

8086 Microprocessor

Microprocessor

Program controlled semiconductor device (IC) which fetches (from memory), decodes and executes instructions.

It is used as CPU (Central Processing Unit) in computers.

Microprocessor

Third Generation

During 1978

HMOS technology \Rightarrow Faster speed

16 bit processors \Rightarrow 40/ 48/ 64 pins

Easier to program

Processor has multiply/ divide arithmetic hardware

More powerful interrupt handling capabilities

Flexible I/O port addressing

Intel 8086 (16 bit processor)

First Generation

Between 1971 – 1973

PMOS technology, non compatible with TTL

4 bit processors \Rightarrow 16 pins

Due to limitations of pins, signals are multiplexed

Fifth Generation **Pentium**

Fourth Generation

During 1980s

Low power version of HMOS technology (HCMOS)

32 bit processors

Physical memory space 2^{24} bytes = 16 Mb

Virtual memory space 2^{40} bytes = 1 Tb

Floating point hardware

Supports increased number of addressing modes

Intel 80386

Second Generation

During 1973

NMOS technology \Rightarrow Faster speed,

Compatible with TTL

8 bit processors \Rightarrow 40 pins

Ability to address large memory spaces and I/O ports

Greater number of levels of subroutine nesting

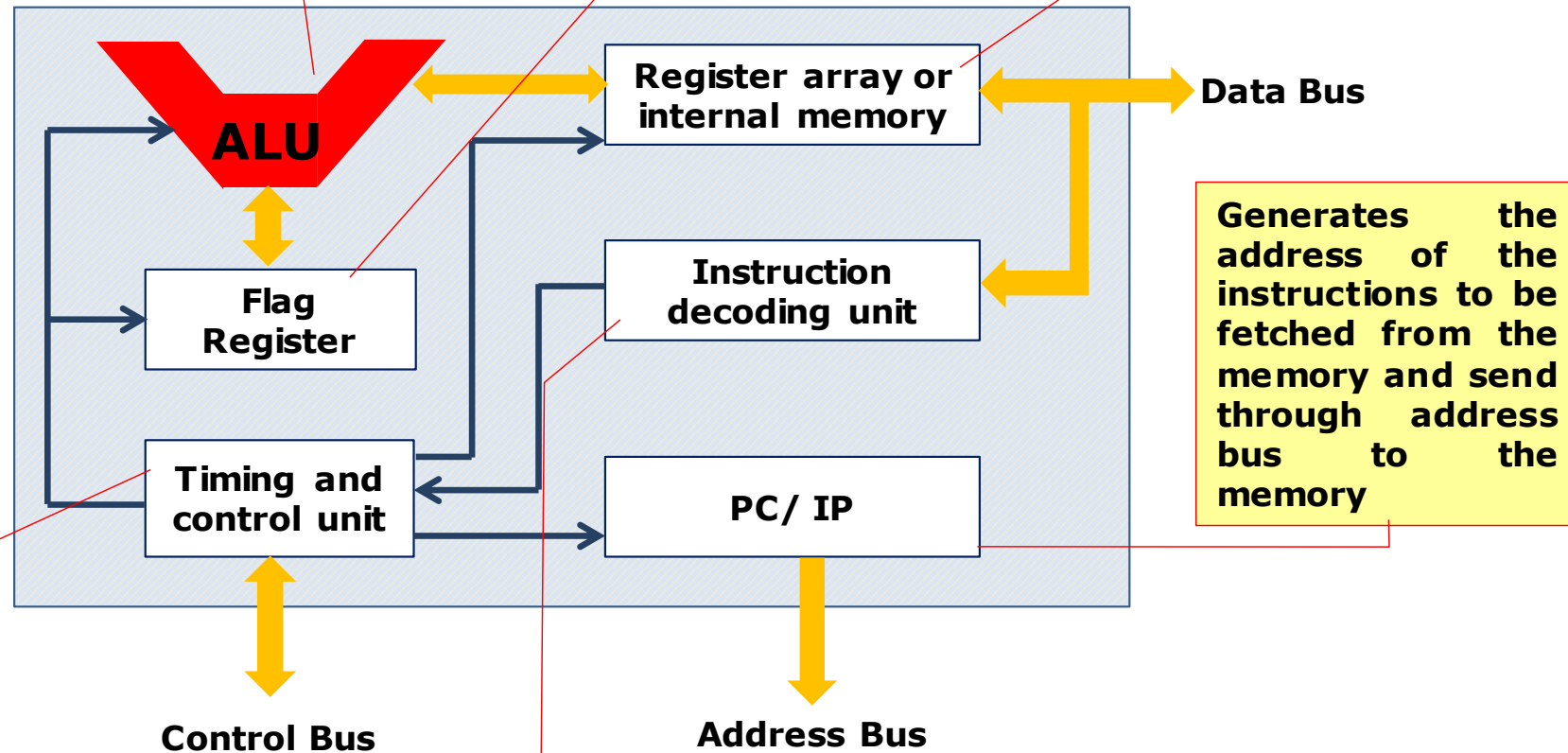
Better interrupt handling capabilities

Intel 8085 (8 bit processor)

Computational Unit;
performs arithmetic and
logic operations

Various conditions of the
results are stored as
status bits called flags in
flag register

Internal storage of data



Generates control signals for
internal and external
operations of the
microprocessor

Decodes instructions; sends
information to the timing and
control unit

Generates the
address of the
instructions to be
fetched from the
memory and send
through address
bus to the
memory

First 16-bit processor released by INTEL in the year 1978

Originally HMOS, now manufactured using HMOS III technique

Approximately 29, 000 transistors, 40 pin, 5V supply

Does not have internal clock; external asymmetric clock source with 33% duty cycle

20-bit address to access memory \Rightarrow can address up to $2^{20} = 1$ megabytes of memory space.

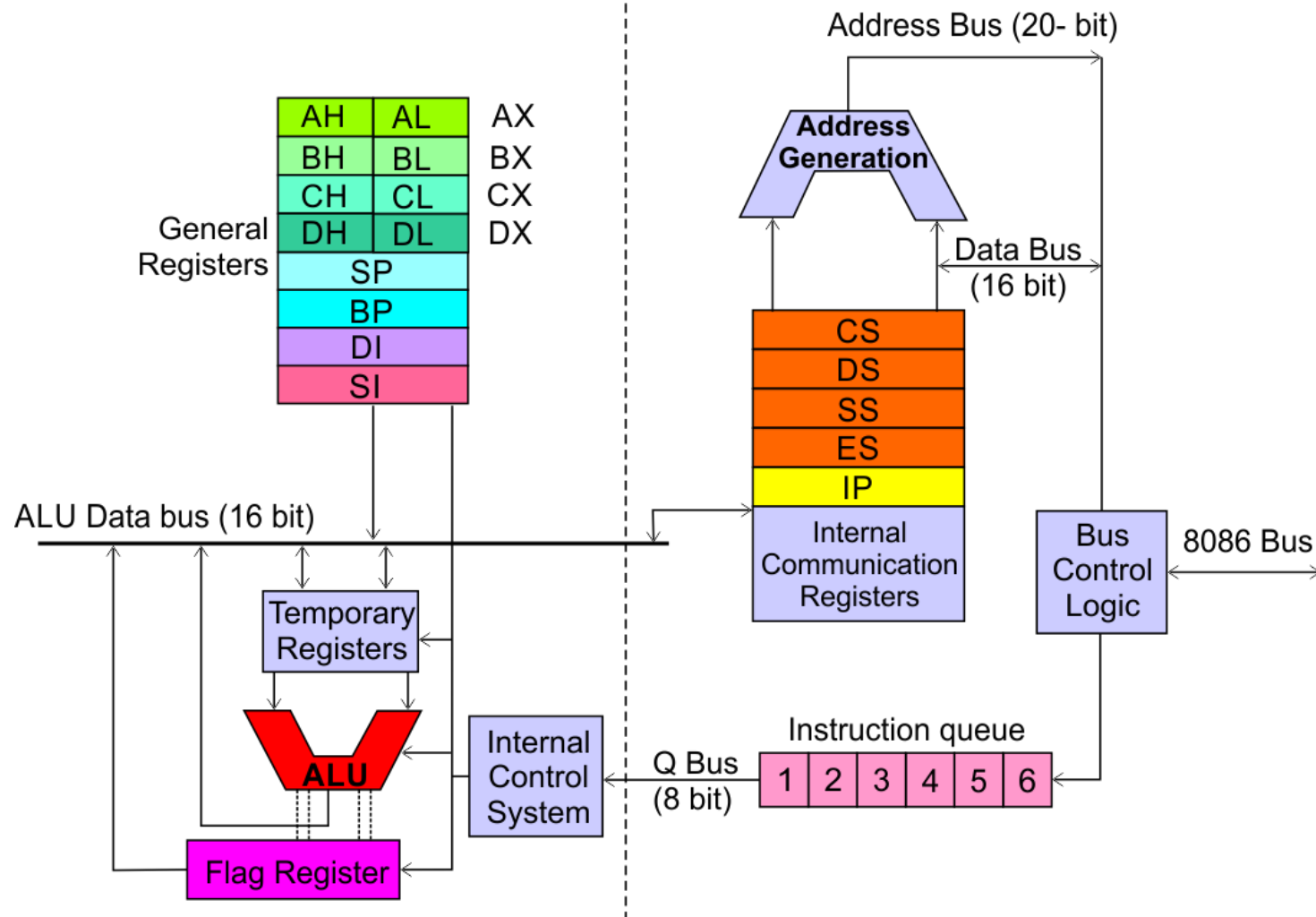
Addressable memory space is organized in to two banks of 512 kb each; **Even (or lower) bank and **Odd (or higher) bank**. Address line A_0 is used to select even bank and control signal \overline{BHE} is used to access odd bank**

Uses a separate 16 bit address for I/O mapped devices \Rightarrow can generate $2^{16} = 64$ k addresses.

Operates in two modes: **minimum mode and **maximum mode**, decided by the signal at MN and \overline{MX} pins.**

Architecture

Architecture



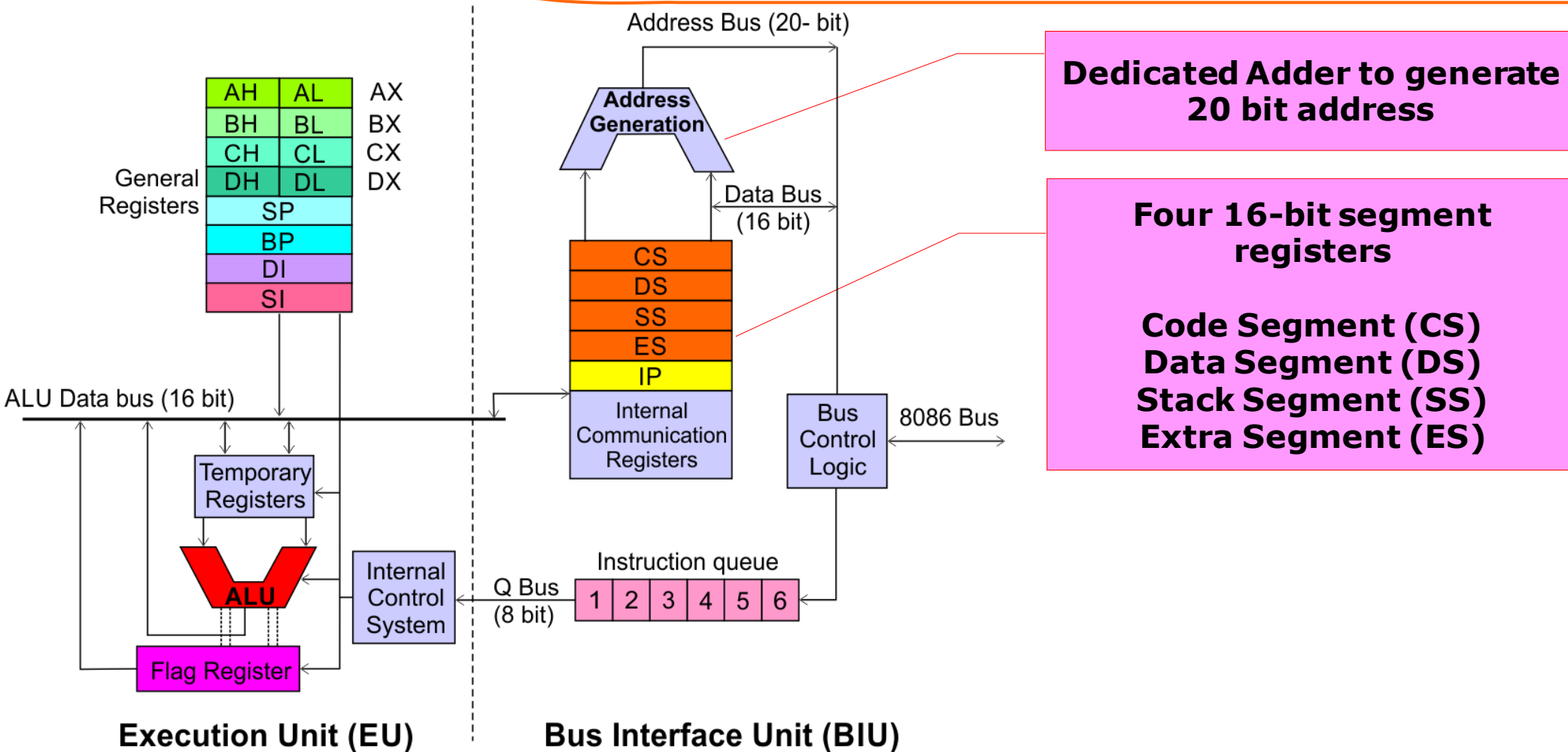
Execution Unit (EU)

EU executes instructions that have already been fetched by the BIU.

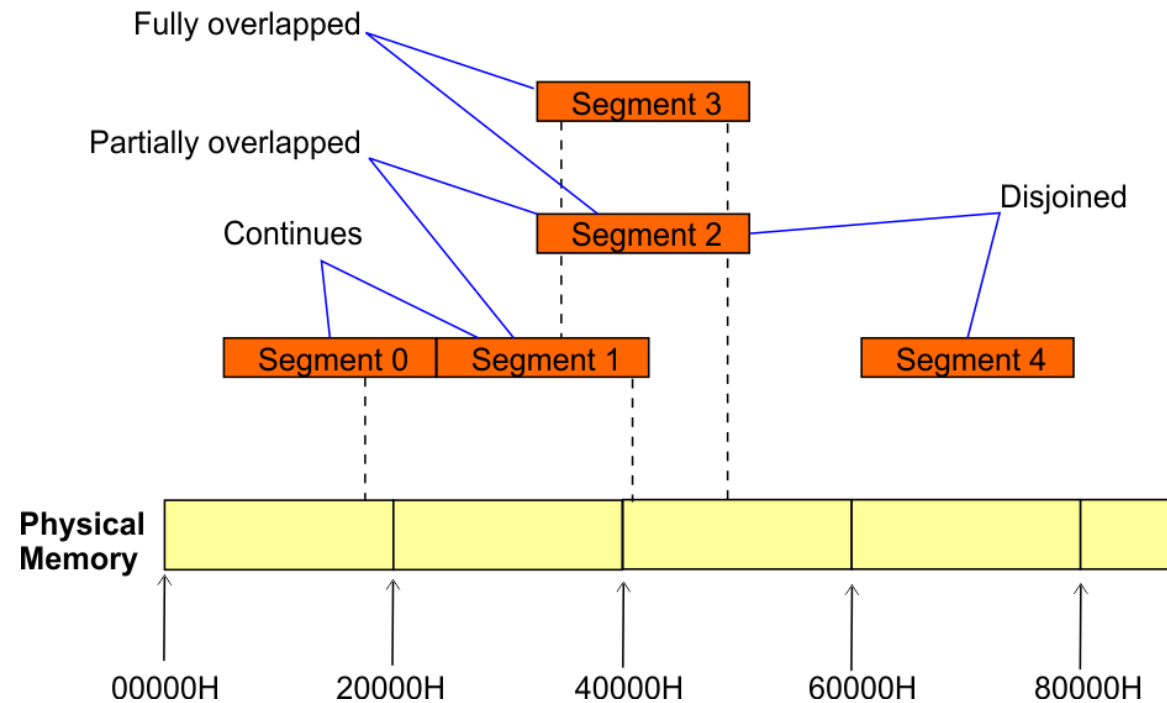
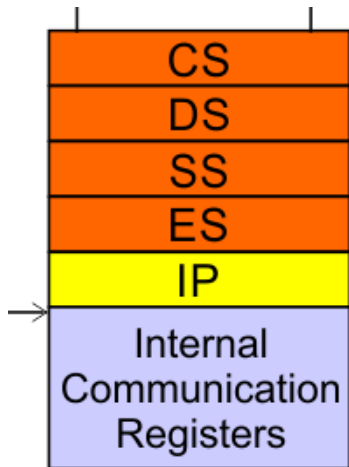
BIU and EU functions separately.

Bus Interface Unit (BIU)

BIU fetches instructions, reads data from memory and I/O ports, writes data to memory and I/O ports.



Segment Registers

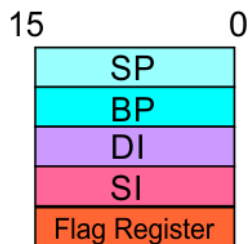
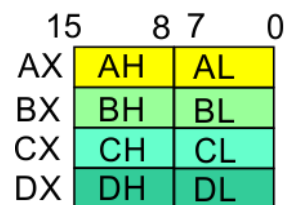


- 8086's 1-megabyte memory is divided into segments of up to 64K bytes each.
- The 8086 can directly address four segments (256 K bytes within the 1 M byte of memory) at a particular time.
- Programs obtain access to code and data in the segments by changing the segment register content to point to the desired segments.

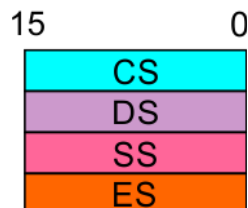
Segment Registers

Code Segment Register

- 16-bit
- CS contains the base or start of the current code segment; IP contains the distance or offset from this address to the next instruction byte to be fetched.
- BIU computes the 20-bit physical address by logically shifting the contents of CS 4-bits to the left and then adding the 16-bit contents of IP.



EU

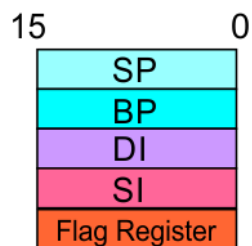
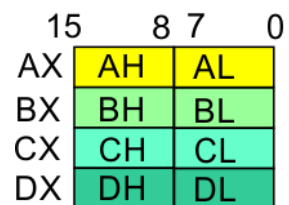
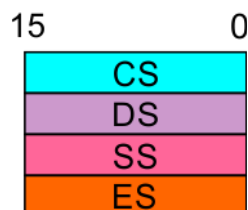


BIU

Segment Registers

Data Segment Register

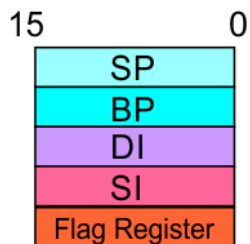
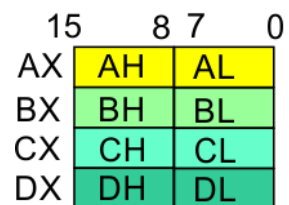
- **16-bit**
- **Points to the current data segment; operands for most instructions are fetched from this segment.**
- **The 16-bit contents of the Source Index (SI) or Destination Index (DI) or a 16-bit displacement are used as offset for computing the 20-bit physical address.**

**EU****BIU**

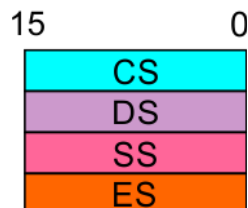
Segment Registers

Stack Segment Register

- 16-bit
- Points to the current stack.
- The 20-bit physical stack address is calculated from the Stack Segment (SS) and the Stack Pointer (SP) for stack instructions such as **PUSH** and **POP**.
- In based addressing mode, the 20-bit physical stack address is calculated from the Stack segment (SS) and the Base Pointer (BP).



EU

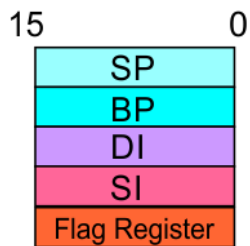
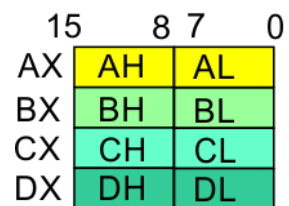
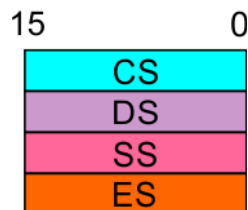


BIU

Segment Registers

Extra Segment Register

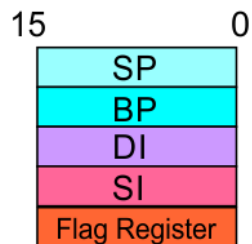
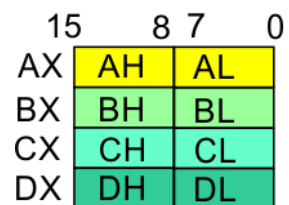
- **16-bit**
- **Points to the extra segment in which data (in excess of 64K pointed to by the DS) is stored.**
- **String instructions use the ES and DI to determine the 20-bit physical address for the destination.**

**EU****BIU**

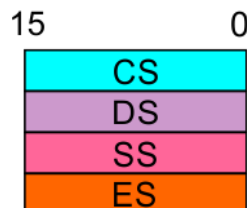
Segment Registers

Instruction Pointer

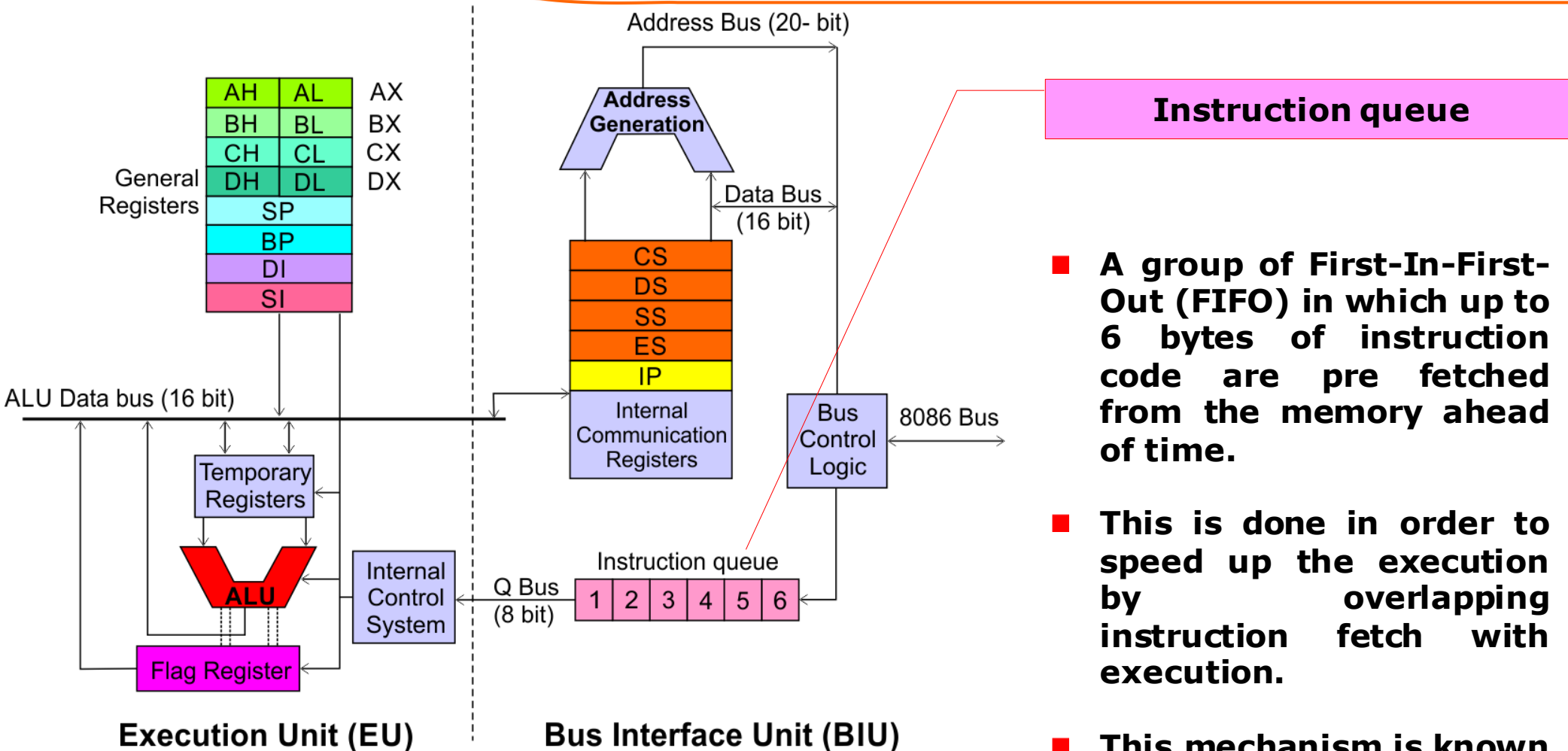
- 16-bit
- Always points to the next instruction to be executed within the currently executing code segment.
- So, this register contains the 16-bit offset address pointing to the next instruction code within the 64Kb of the code segment area.
- Its content is automatically incremented as the execution of the next instruction takes place.



EU



BIU



- A group of First-In-First-Out (FIFO) in which up to 6 bytes of instruction code are pre fetched from the memory ahead of time.
- This is done in order to speed up the execution by overlapping instruction fetch with execution.
- This mechanism is known as pipelining.

EU decodes and executes instructions.

A decoder in the EU control system translates instructions.

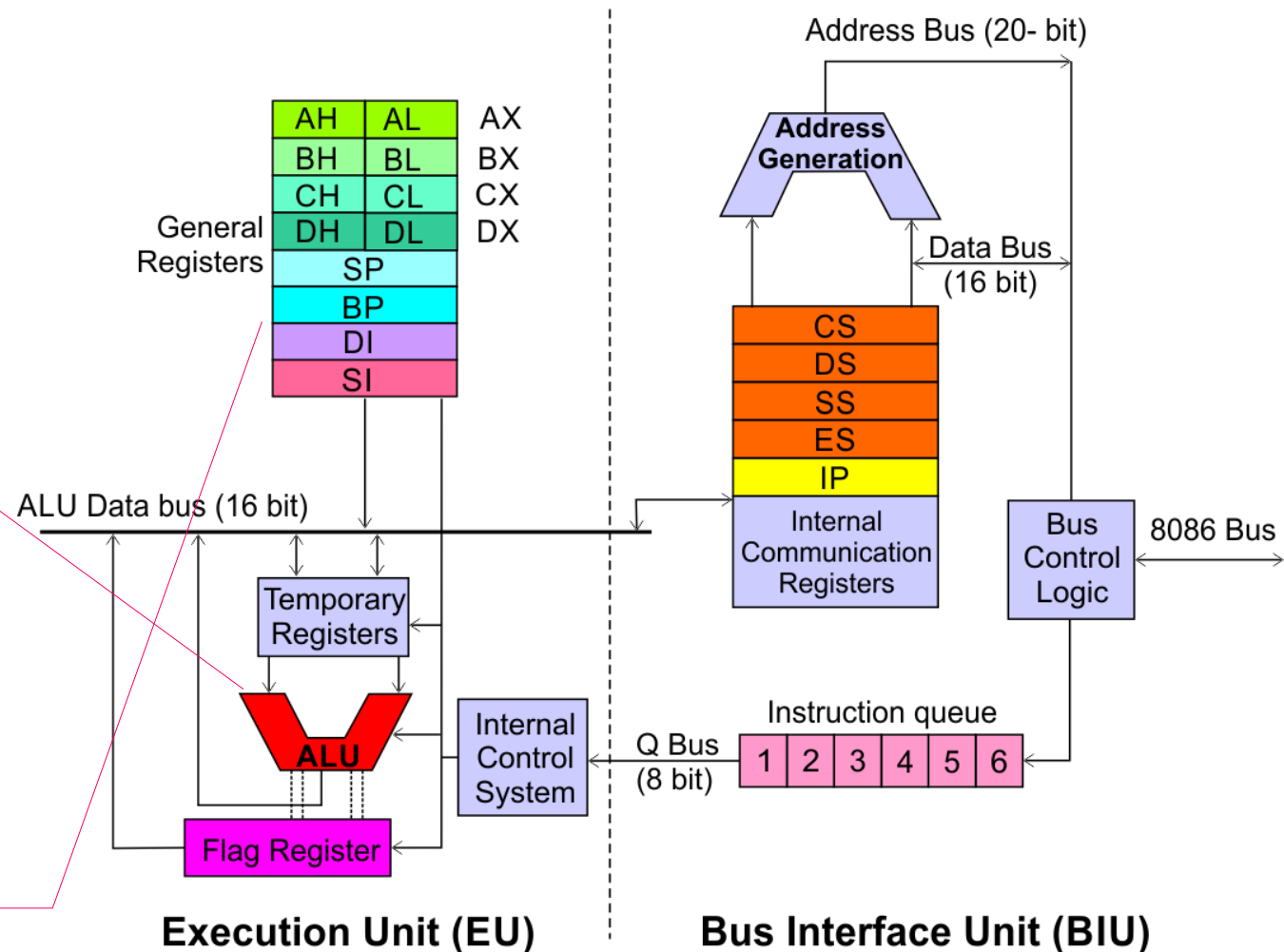
16-bit ALU for performing arithmetic and logic operation

Four general purpose registers (AX, BX, CX, DX);

Pointer registers (Stack Pointer, Base Pointer);

and

Index registers (Source Index, Destination Index) each of 16-bits



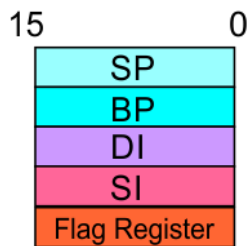
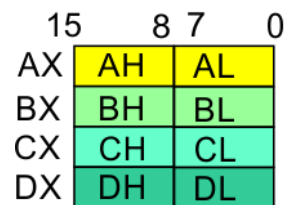
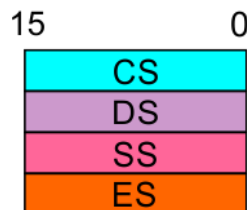
Some of the 16 bit registers can be used as two 8 bit registers as :

AX can be used as AH and AL
 BX can be used as BH and BL
 CX can be used as CH and CL
 DX can be used as DH and DL

EU Registers

Accumulator Register (AX)

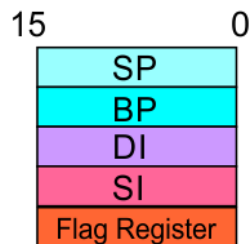
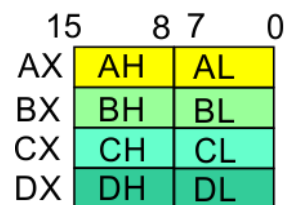
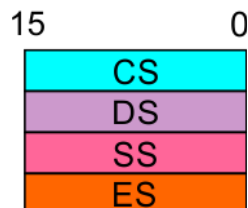
- Consists of two 8-bit registers AL and AH, which can be combined together and used as a 16-bit register AX.
- AL in this case contains the low order byte of the word, and AH contains the high-order byte.

**EU****BIU**

EU Registers

Base Register (BX)

- Consists of two 8-bit registers BL and BH, which can be combined together and used as a 16-bit register BX.
- BL in this case contains the low-order byte of the word, and BH contains the high-order byte.
- This is the only general purpose register whose contents can be used for addressing the 8086 memory.

**EU****BIU**

EU Registers

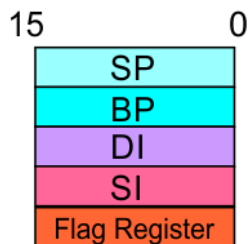
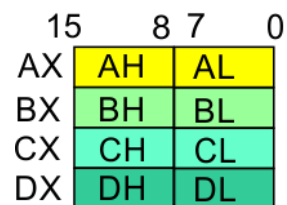
Counter Register (CX)

- Consists of two 8-bit registers CL and CH, which can be combined together and used as a 16-bit register CX.
- When combined, CL register contains the low order byte of the word, and CH contains the high-order byte.
- Instructions such as **SHIFT**, **ROTATE** and **LOOP** use the contents of CX as a counter.

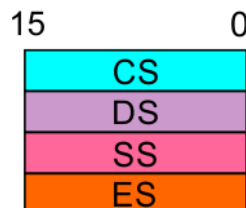
Example:

The instruction **LOOP START** automatically decrements CX by 1 without affecting flags and will check if [CX] = 0.

If it is zero, 8086 executes the next instruction; otherwise the 8086 branches to the label **START**.



EU

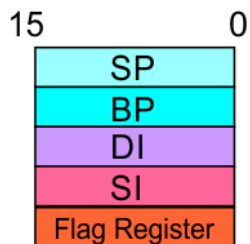
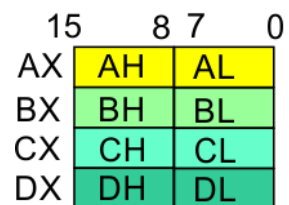
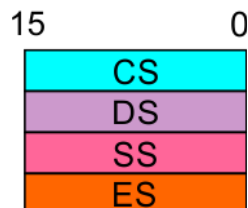


BIU

EU Registers

Data Register (DX)

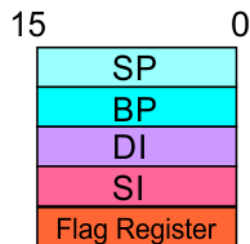
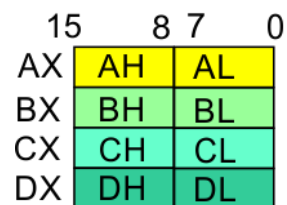
- Consists of two 8-bit registers DL and DH, which can be combined together and used as a 16-bit register DX.
- When combined, DL register contains the low order byte of the word, and DH contains the high-order byte.

**EU****BIU**

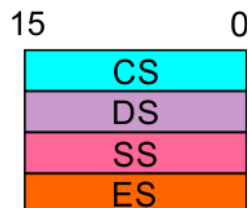
EU Registers

Stack Pointer (SP) and Base Pointer (BP)

- SP and BP are used to access data in the stack segment.
- SP is used as an offset from the current SS during execution of instructions that involve the stack segment in the external memory.
- SP contents are automatically updated (incremented/decremented) due to execution of a POP or PUSH instruction.
- BP contains an offset address in the current SS, which is used by instructions utilizing the based addressing mode.



EU

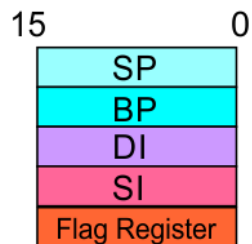
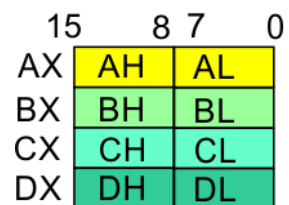
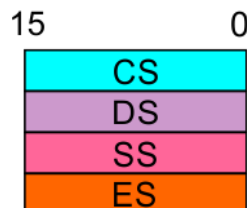


BIU

EU Registers

Source Index (SI) and Destination Index (DI)

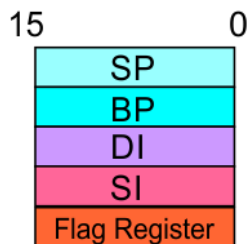
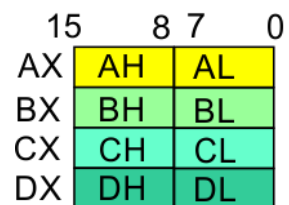
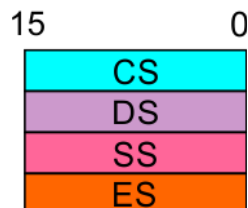
- Used in indexed addressing.
- Instructions that process data strings use the SI and DI registers together with DS and ES respectively in order to distinguish between the source and destination addresses.

**EU****BIU**

EU Registers

Source Index (SI) and Destination Index (DI)

- Used in indexed addressing.
- Instructions that process data strings use the SI and DI registers together with DS and ES respectively in order to distinguish between the source and destination addresses.

**EU****BIU**

Flag Register

Auxiliary Carry Flag

This is set, if there is a carry from the lowest nibble, i.e, bit three during addition, or borrow for the lowest nibble, i.e, bit three, during subtraction.

Carry Flag

This flag is set, when there is a carry out of MSB in case of addition or a borrow in case of subtraction.

Sign Flag

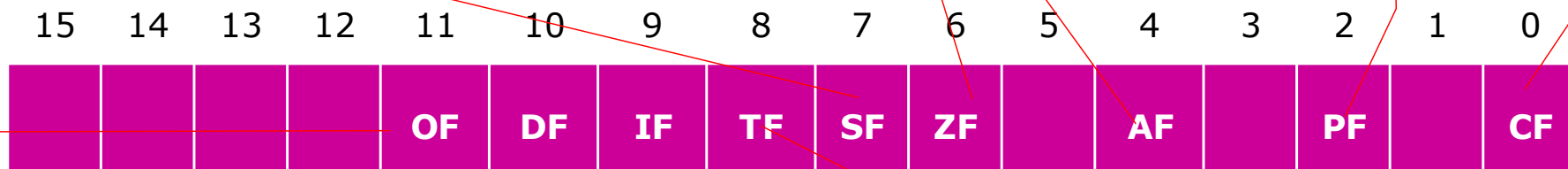
This flag is set, when the result of any computation is negative

Zero Flag

This flag is set, if the result of the computation or comparison performed by an instruction is zero

Parity Flag

This flag is set to 1, if the lower byte of the result contains even number of 1's ; for odd number of 1's set to zero.



Over flow Flag

This flag is set, if an overflow occurs, i.e, if the result of a signed operation is large enough to accommodate in a destination register. The result is of more than 7-bits in size in case of 8-bit signed operation and more than 15-bits in size in case of 16-bit sign operations, then the overflow will be set.

Trap Flag

If this flag is set, the CPU automatically generates an internal interrupt after each instruction, allowing a program to be inspected as it executes instruction by instruction.

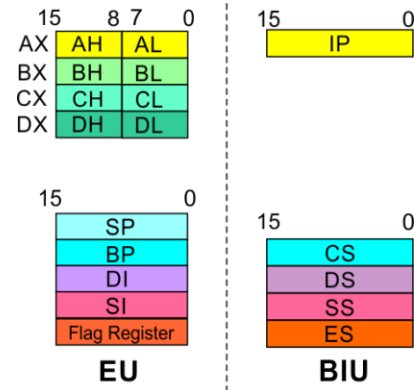
Direction Flag

This is used by string manipulation instructions. If this flag bit is '0', the string is processed beginning from the lowest address to the highest address, i.e., auto incrementing mode. Otherwise, the string is processed from the highest address towards the lowest address, i.e., auto decrementing mode.

Interrupt Flag

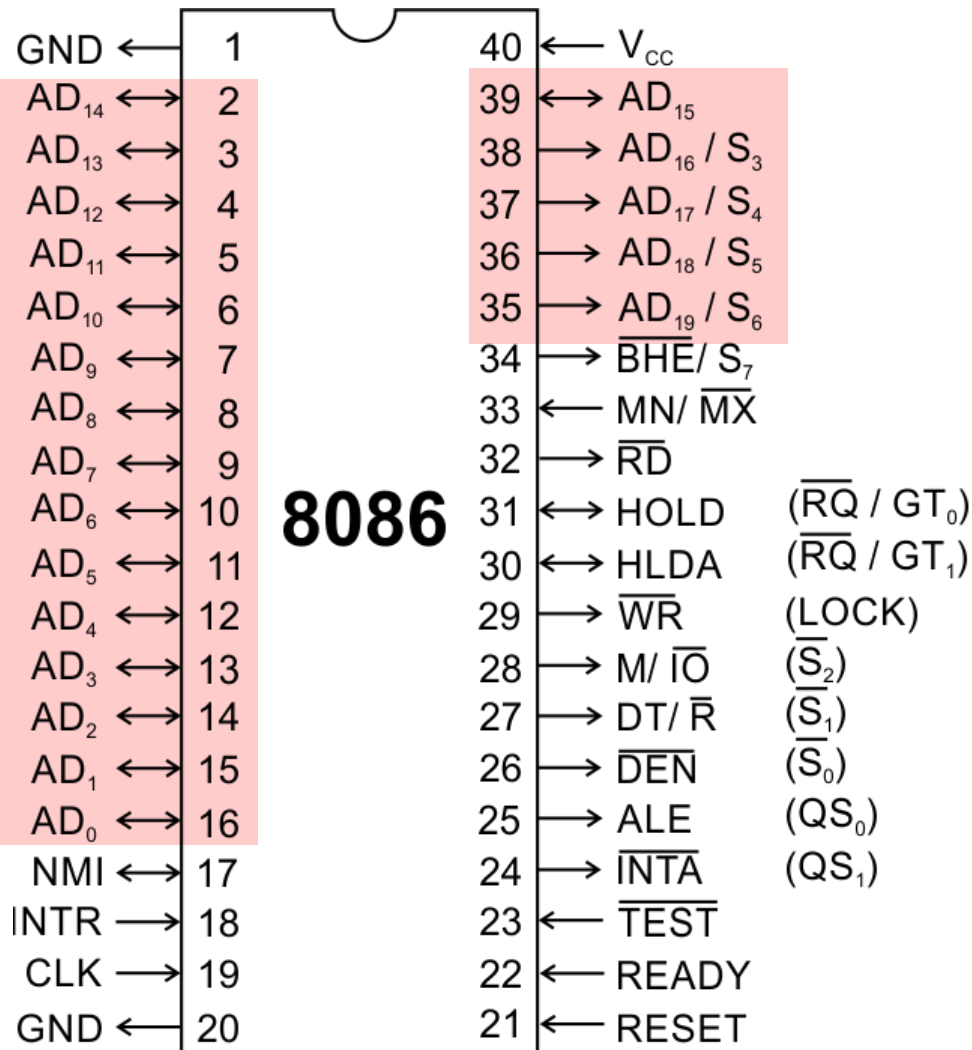
Causes the 8086 to recognize external mask interrupts; clearing IF disables these interrupts.

**8086 registers
categorized
into 4 groups**



Sl.No.	Type	Register width	Name of register
1	General purpose register	16 bit	AX, BX, CX, DX
		8 bit	AL, AH, BL, BH, CL, CH, DL, DH
2	Pointer register	16 bit	SP, BP
3	Index register	16 bit	SI, DI
4	Instruction Pointer	16 bit	IP
5	Segment register	16 bit	CS, DS, SS, ES
6	Flag (PSW)	16 bit	Flag register

Pins and signals

8086 **AD_0-AD_{15} (Bidirectional)****Address/Data bus**

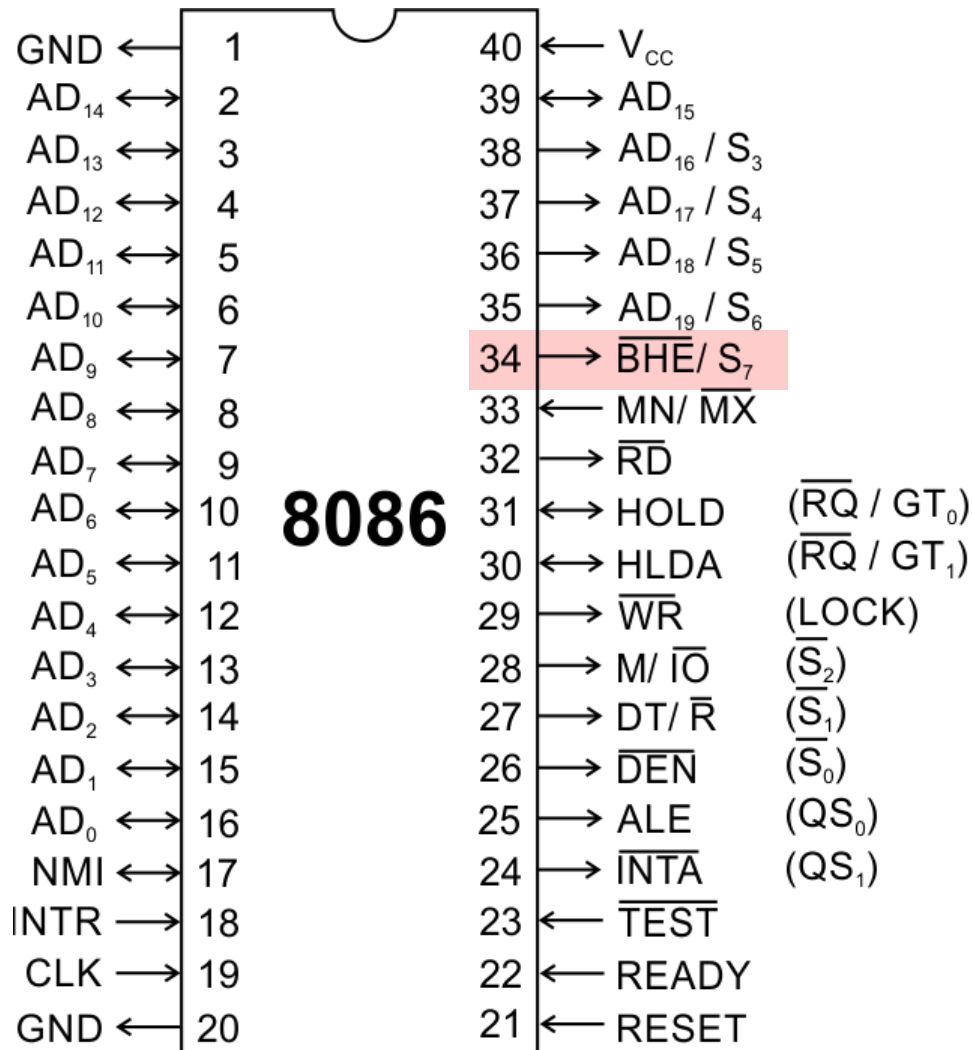
Low order address bus; these are multiplexed with data.

When AD lines are used to transmit memory address the symbol A is used instead of AD, for example A_0-A_{15} .

When data are transmitted over AD lines the symbol D is used in place of AD, for example D_0-D_7 , D_8-D_{15} or D_0-D_{15} .

 $A_{16}/S_3, A_{17}/S_4, A_{18}/S_5, A_{19}/S_6$

High order address bus. These are multiplexed with status signals



BHE (Active Low)/ S_7 (Output)

Bus High Enable/Status

It is used to enable data onto the most significant half of data bus, D_8 - D_{15} . 8-bit device connected to upper half of the data bus use BHE (Active Low) signal. It is multiplexed with status signal S_7 .

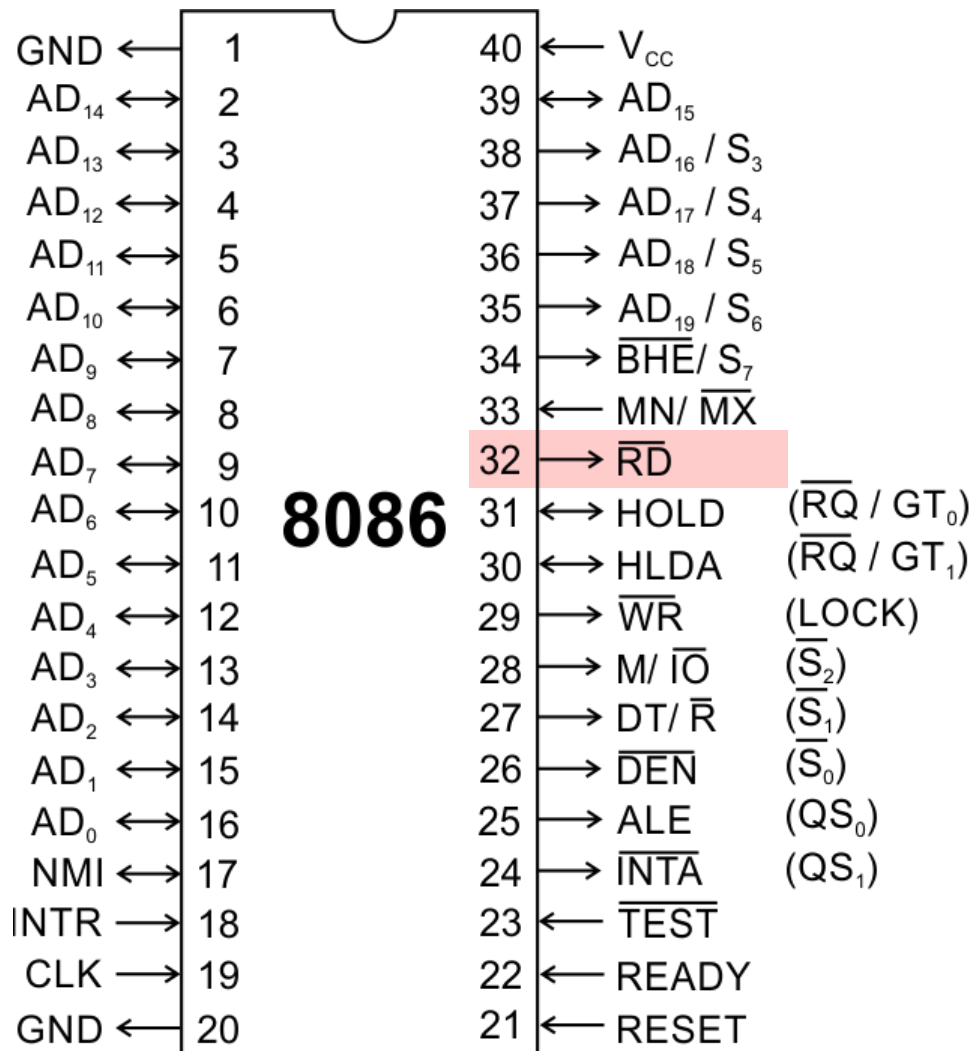
MN/ MX

MINIMUM / MAXIMUM

This pin signal indicates what mode the processor is to operate in.

RD (Read) (Active Low)

The signal is used for read operation.
It is an output signal.
It is active when low.



TEST

\overline{TEST} input is tested by the 'WAIT' instruction.

8086 will enter a wait state after execution of the WAIT instruction and will resume execution only when the \overline{TEST} is made low by an active hardware.

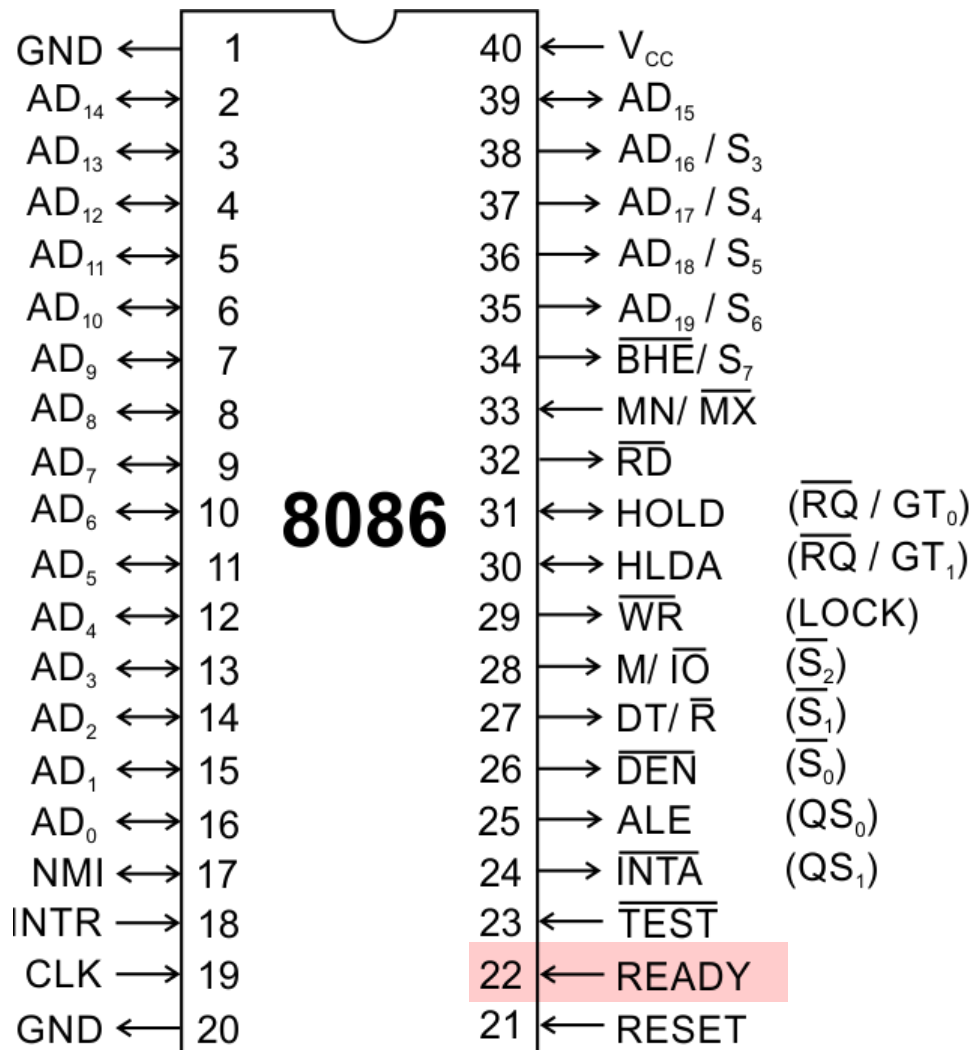
This is used to synchronize an external activity to the processor internal operation.

READY

This is the acknowledgement from the slow device or memory that they have completed the data transfer.

The signal made available by the devices is synchronized by the 8284A clock generator to provide ready input to the **8086**.

The signal is active high.



RESET (Input)

Causes the processor to immediately terminate its present activity.

The signal must be active HIGH for at least four clock cycles.

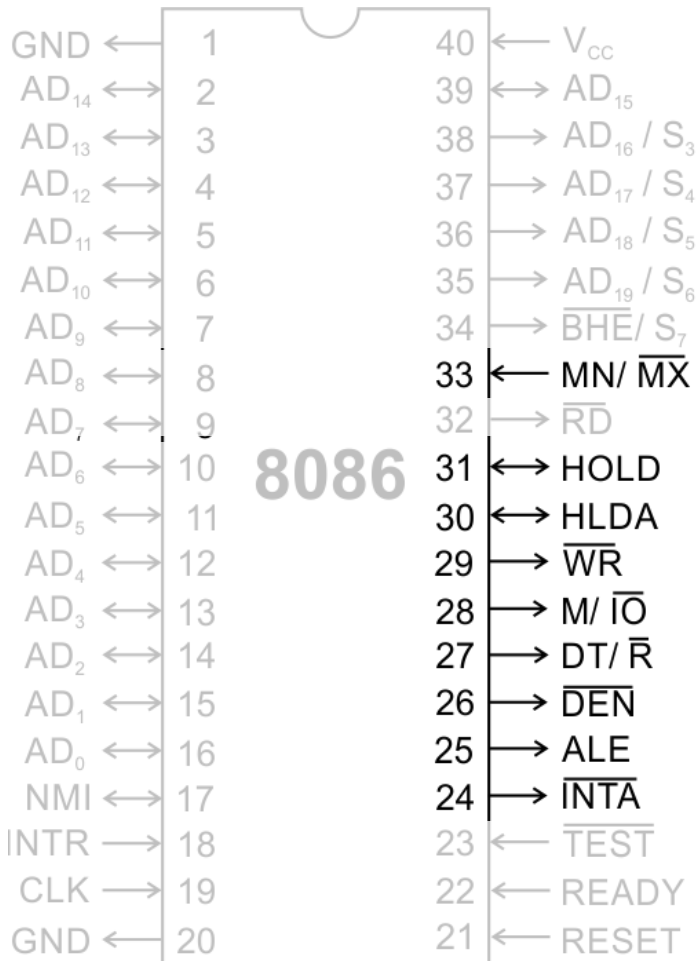
CLK

The clock input provides the basic timing for processor operation and bus control activity. Its an asymmetric square wave with 33% duty cycle.

INTR Interrupt Request

This is a triggered input. This is sampled during the last clock cycles of each instruction to determine the availability of the request. If any interrupt request is pending, the processor enters the interrupt acknowledge cycle.

This signal is active high and internally synchronized.



The 8086 microprocessor can work in two modes of operations : **Minimum mode** and **Maximum mode**.

In the minimum mode of operation the microprocessor do not associate with any co-processors and can not be used for multiprocessor systems.

In the maximum mode the 8086 can work in multi-processor or co-processor configuration.

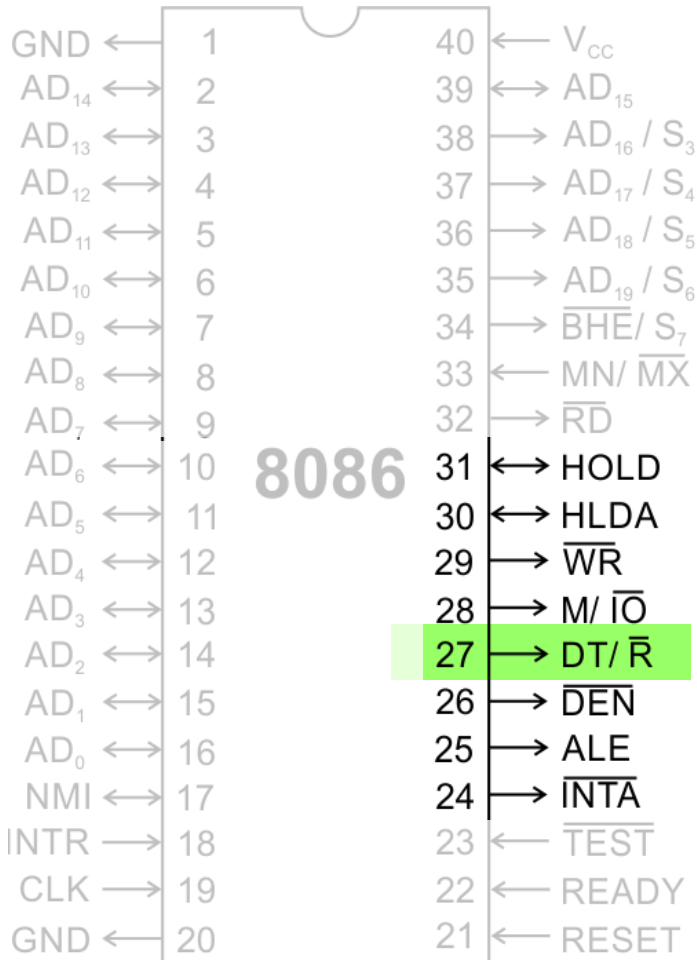
Minimum or maximum mode operations are decided by the pin MN / \overline{MX} (Active low).

When this pin is high 8086 operates in minimum mode otherwise it operates in Maximum mode.

Pins 24 -31

For minimum mode operation, the $\overline{MN}/\overline{MX}$ is tied to VCC (logic high)

8086 itself generates all the bus control signals

**DT/ \overline{R}**

(**Data Transmit/ Receive**) Output signal from the processor to control the direction of data flow through the data transceivers

 \overline{DEN}

(**Data Enable**) It is used to enable Transreceiver 8286. The transreceiver is a device used to separate data from the address/data bus.

ALE

(**Address Latch Enable**) Used to demultiplex the address and data lines using external latches

M/ \overline{IO}

Used to differentiate memory access and I/O access. For memory reference instructions, it is **high**. For IN and OUT instructions, it is **low**.

 \overline{WR}

Write control signal; asserted **low** Whenever processor writes data to memory or I/O port

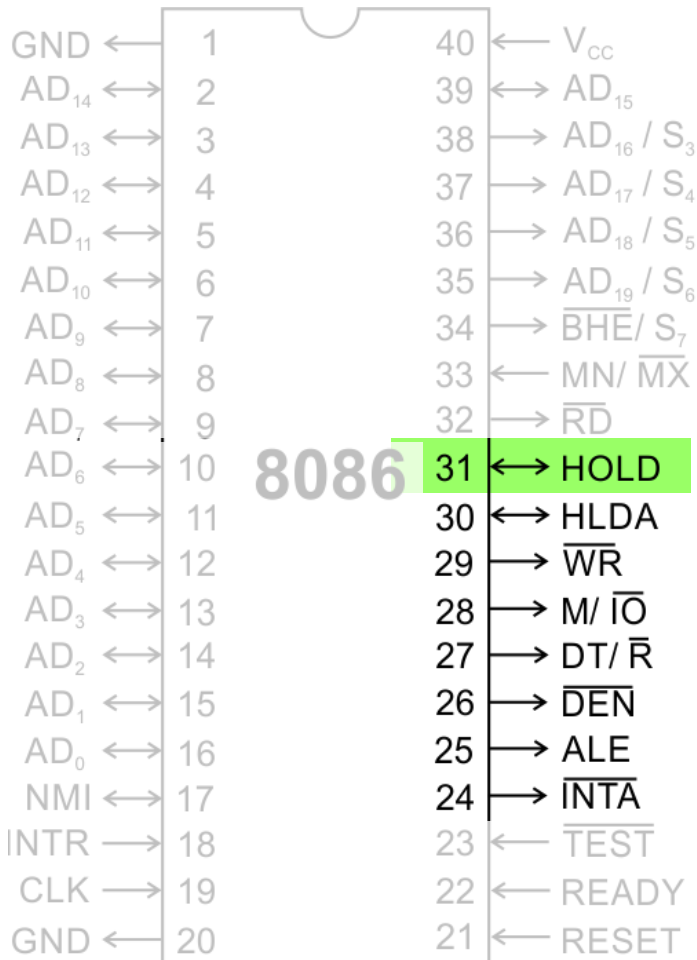
 \overline{INTA}

(**Interrupt Acknowledge**) When the interrupt request is accepted by the processor, the output is **low** on this line.

Pins 24 -31

For minimum mode operation, the $\overline{MN}/\overline{MX}$ is tied to VCC (logic high)

8086 itself generates all the bus control signals

**HOLD**

Input signal to the processor from the bus masters as a request to grant the control of the bus.

Usually used by the DMA controller to get the control of the bus.

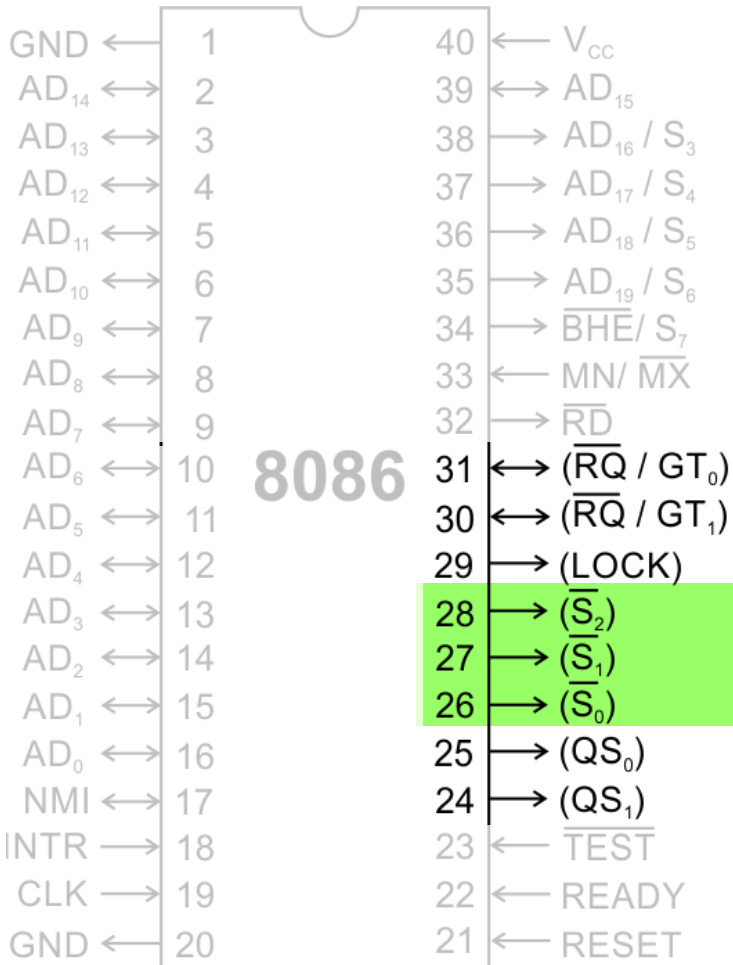
HLDA

(**Hold Acknowledge**) Acknowledge signal by the processor to the bus master requesting the control of the bus through HOLD.

The acknowledge is asserted high, when the processor accepts HOLD.

During maximum mode operation, the $\overline{MN}/\overline{MX}$ is grounded (logic low)

Pins 24 -31 are reassigned



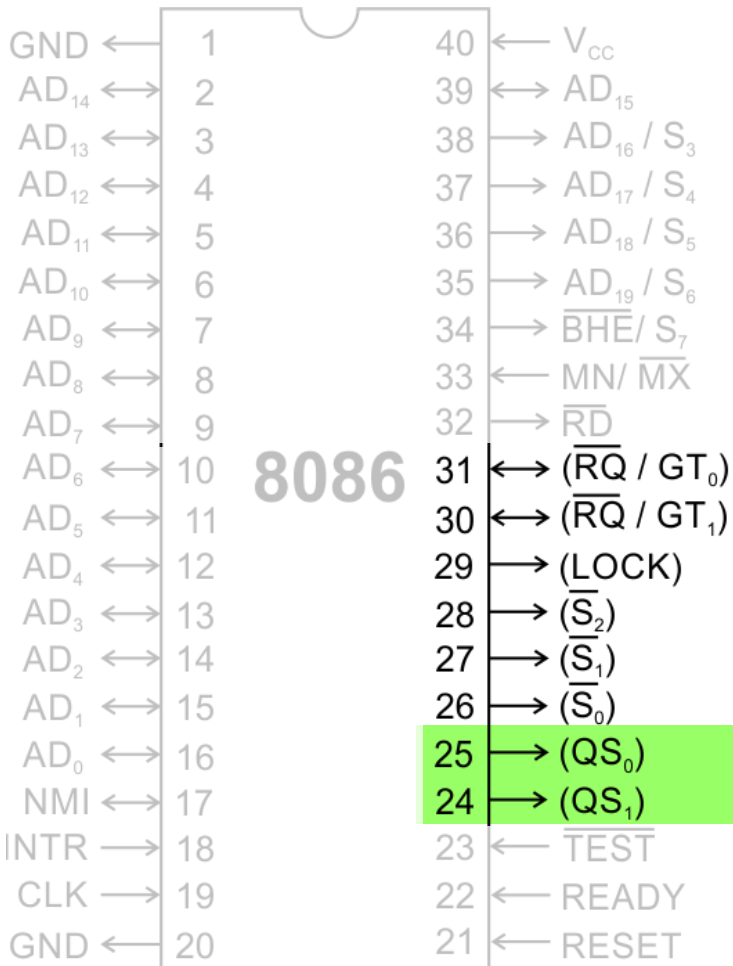
$\overline{S_0}, \overline{S_1}, \overline{S_2}$

Status signals; used by the 8086 bus controller to generate bus timing and control signals. These are decoded as shown.

Status Signal			Machine Cycle
$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	
0	0	0	Interrupt acknowledge
0	0	1	Read I/O port
0	1	0	Write I/O port
0	1	1	Halt
1	0	0	Code access
1	0	1	Read memory
1	1	0	Write memory
1	1	1	Passive/Inactive

During maximum mode operation, the $\overline{MN}/\overline{MX}$ is grounded (logic low)

Pins 24 -31 are reassigned



$\overline{QS_0}, \overline{QS_1}$

(Queue Status) The processor provides the status of queue in these lines.

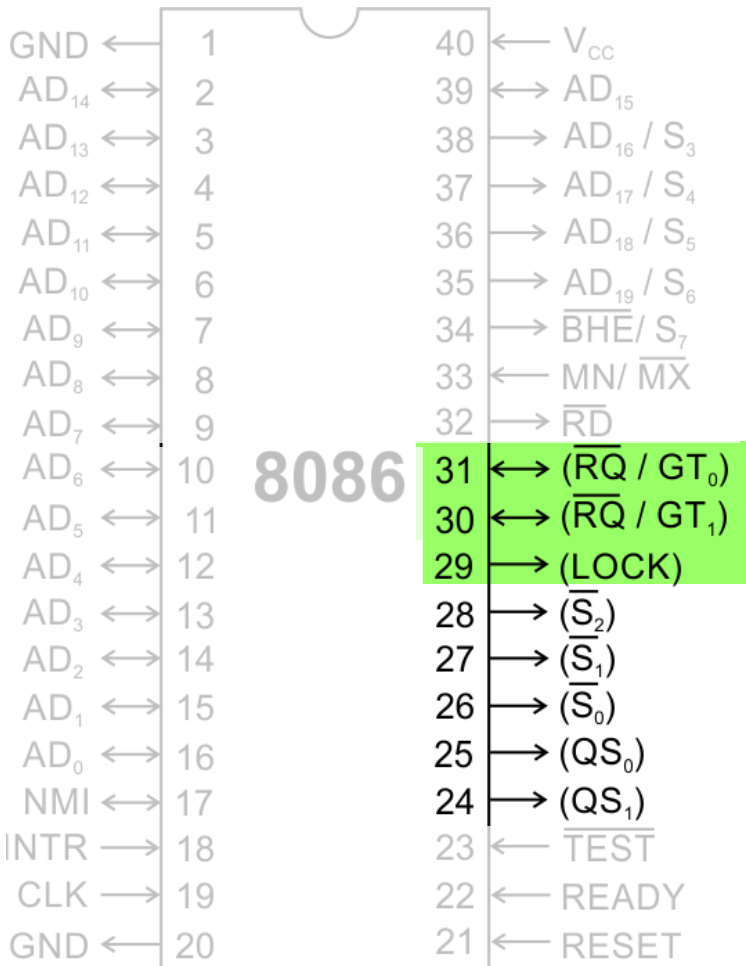
The queue status can be used by external device to track the internal status of the queue in 8086.

The output on QS_0 and QS_1 can be interpreted as shown in the table.

Queue status		Queue operation
QS_1	QS_0	
0	0	No operation
0	1	First byte of an opcode from queue
1	0	Empty the queue
1	1	Subsequent byte from queue

During maximum mode operation, the $\overline{MN}/\overline{MX}$ is grounded (logic low)

Pins 24 -31 are reassigned



\overline{RQ}/GT_0 ,
 \overline{RQ}/GT_1

(Bus Request/ Bus Grant) These requests are used by other local bus masters to force the processor to release the local bus at the end of the processor's current bus cycle.

These pins are bidirectional.

The request on $\overline{GT_0}$ will have higher priority than $\overline{GT_1}$

\overline{LOCK}

An output signal activated by the LOCK prefix instruction.

Remains active until the completion of the instruction prefixed by LOCK.

The 8086 output low on the \overline{LOCK} pin while executing an instruction prefixed by LOCK to prevent other bus masters from gaining control of the system bus.

ADDRESSING MODES & Instruction set

Program is a set of instructions written to solve a problem. Instructions are the directions which a microprocessor follows to execute a task or part of a task.

Broadly, computer language can be divided into two parts as high-level language and low level language. Low level language are machine specific. Low level language can be further divided into machine language and assembly language.

Machine language is the only language which a machine can understand. Instructions in this language are written in binary bits as a specific bit pattern. The computer interprets this bit pattern as an instruction to perform a particular task. The entire program is a sequence of binary numbers. This is a machine-friendly language but not user friendly. Debugging is another problem associated with machine language.

To overcome these problems, programmers develop another way in which instructions are written in English alphabets. This new language is known as Assembly language. The instructions in this language are termed *mnemonics*. As microprocessor can only understand the machine language so mnemonics are translated into machine language either manually or by a program known as *assembler*.

Efficient software development for the microprocessor requires a complete familiarity with the instruction set, their format and addressing modes. Here in this chapter, we will focus on the addressing modes and instructions formats of microprocessor 8086.

ADDRESSING MODES

Addressing Modes

- Every instruction of a program has to operate on a data.
- The different ways in which a source operand is denoted in an instruction are known as addressing modes.

1. Register Addressing

2. Immediate Addressing

Group I : Addressing modes for register and immediate data

3. Direct Addressing

4. Register Indirect Addressing

5. Based Addressing

6. Indexed Addressing

7. Based Index Addressing

8. String Addressing

Group II : Addressing modes for memory data

9. Direct I/O port Addressing

10. Indirect I/O port Addressing

Group III : Addressing modes for I/O ports

11. Relative Addressing

Group IV : Relative Addressing mode

12. Implied Addressing

Group V : Implied Addressing mode

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing

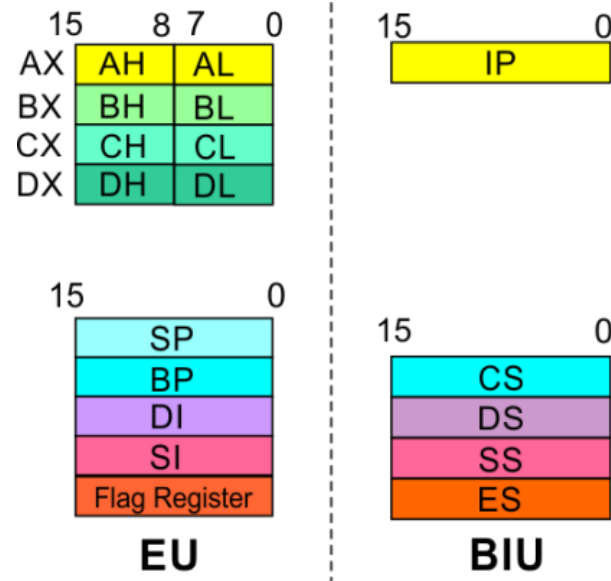
The instruction will specify the name of the register which holds the data to be operated by the instruction.

Example:

MOV CL, DH

The content of 8-bit register DH is moved to another 8-bit register CL

$(CL) \leftarrow (DH)$



1. Register Addressing

2. Immediate Addressing

3. Direct Addressing

4. Register Indirect Addressing

5. Based Addressing

6. Indexed Addressing

7. Based Index Addressing

8. String Addressing

9. Direct I/O port Addressing

10. Indirect I/O port Addressing

11. Relative Addressing

12. Implied Addressing

In immediate addressing mode, an 8-bit or 16-bit data is specified as part of the instruction

Example:

MOV DL, 08H

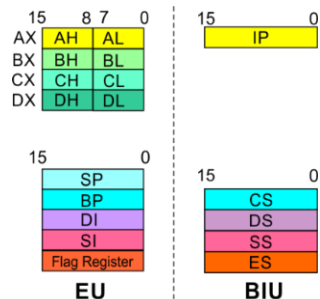
The 8-bit data (08_H) given in the instruction is moved to DL

$(DL) \leftarrow 08_H$

MOV AX, 0A9FH

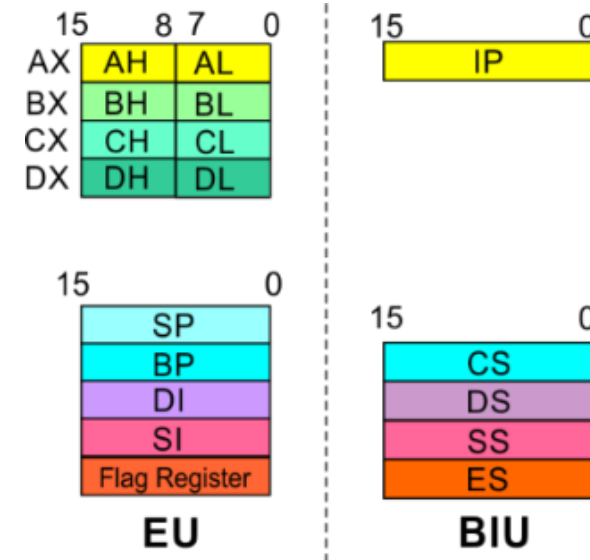
The 16-bit data (0A9F_H) given in the instruction is moved to AX register

$(AX) \leftarrow 0A9F_H$



Addressing Modes : Memory Access

- 20 Address lines \Rightarrow 8086 can address up to $2^{20} = 1\text{M}$ bytes of memory
- However, the largest register is only 16 bits
- Physical Address will have to be calculated
Physical Address : Actual address of a byte in memory. i.e. the value which goes out onto the address bus.
- Memory Address represented in the form –
Seg : Offset (Eg - 89AB:F012)
- Each time the processor wants to access memory, it takes the contents of a segment register, shifts it one hexadecimal place to the left (same as multiplying by 16_{10}), then add the required offset to form the 20-bit address



16 bytes of contiguous memory

89AB : F012 \rightarrow 89AB \rightarrow 89AB0 (Paragraph to byte $\rightarrow 89AB \times 10 = 89AB0$)
F012 \rightarrow 0F012 (Offset is already in byte unit)
+ -----
98AC2 (The absolute address)

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing

Here, the effective address of the memory location at which the data operand is stored is given in the instruction.

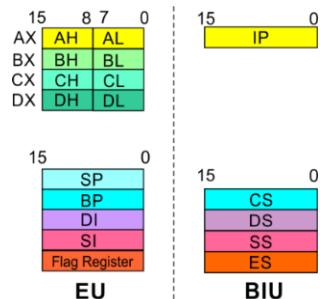
The effective address is just a 16-bit number written directly in the instruction.

Example:

```
MOV BX, [1354H]
MOV BL, [0400H]
```

The square brackets around the 1354_H denotes the contents of the memory location. When executed, this instruction will copy the contents of the memory location into BX register.

This addressing mode is called direct because the displacement of the operand from the segment base is specified directly in the instruction.



1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing

In Register indirect addressing, name of the register which holds the **effective address (EA)** will be specified in the instruction.

Registers used to hold EA are any of the following registers:

BX, BP, DI and SI.

Content of the **DS register** is used for **base address calculation.**

Example:

MOV CX, [BX]

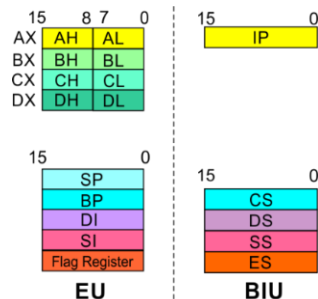
Operations:

$EA = (BX)$
 $BA = (DS) \times 16_{10}$
 $MA = BA + EA$

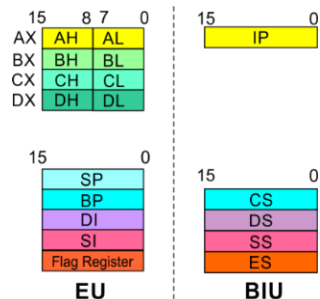
$(CX) \leftarrow (MA) \text{ or,}$

$(CL) \leftarrow (MA)$
 $(CH) \leftarrow (MA + 1)$

Note : Register/ memory enclosed in brackets refer to content of register/ memory



1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



In Based Addressing, **BX or BP** is used to hold the base value for effective address and a **signed 8-bit or unsigned 16-bit** displacement will be specified in the instruction.

In case of 8-bit displacement, it is **sign extended** to 16-bit before adding to the base value.

When **BX** holds the base value of EA, 20-bit physical address is calculated from **BX and DS**.

When **BP** holds the base value of EA, **BP and SS** is used.

Example:

MOV AX, [BX + 08H]

Operations:

$0008_H \leftarrow 08_H$ (Sign extended)

$EA = (BX) + 0008_H$

$BA = (DS) \times 16_{10}$

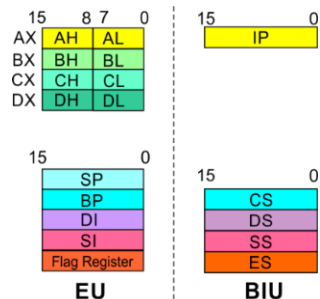
$MA = BA + EA$

$(AX) \leftarrow (MA)$ or,

$(AL) \leftarrow (MA)$

$(AH) \leftarrow (MA + 1)$

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



SI or DI register is used to hold an index value for memory data and a signed 8-bit or unsigned 16-bit displacement will be specified in the instruction.

Displacement is added to the index value in SI or DI register to obtain the EA.

In case of 8-bit displacement, it is sign extended to 16-bit before adding to the base value.

Example:

MOV CX, [SI + 0A2H]

Operations:

$FFA2_H \leftarrow A2_H$ (Sign extended)

$EA = (SI) + FFA2_H$

$BA = (DS) \times 16_{10}$

$MA = BA + EA$

$(CX) \leftarrow (MA)$ or,

$(CL) \leftarrow (MA)$

$(CH) \leftarrow (MA + 1)$

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing

In Based Index Addressing, the effective address is computed from the sum of a base register (BX or BP), an index register (SI or DI) and a displacement.

Example:

MOV DX, [BX + SI + 0AH]

Operations:

$000A_H \leftarrow 0A_H$ (Sign extended)

$EA = (BX) + (SI) + 000A_H$

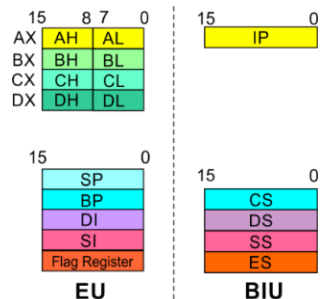
$BA = (DS) \times 16_{10}$

$MA = BA + EA$

$(DX) \leftarrow (MA)$ or,

$(DL) \leftarrow (MA)$

$(DH) \leftarrow (MA + 1)$



1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing

Note : Effective address of the Extra segment register

Employed in string operations to operate on string data.

The effective address (EA) of source data is stored in SI register and the EA of destination is stored in DI register.

Segment register for calculating base address of source data is DS and that of the destination data is ES

Example: MOVSB

Operations:

Calculation of source memory location:

$$EA = (SI) \quad BA = (DS) \times 16_{10} \quad MA = BA + EA$$

Calculation of destination memory location:

$$EA_E = (DI) \quad BA_E = (ES) \times 16_{10} \quad MA_E = BA_E + EA_E$$

$$(MAE) \leftarrow (MA)$$

If $DF = 1$, then $(SI) \leftarrow (SI) - 1$ and $(DI) \leftarrow (DI) - 1$

If $DF = 0$, then $(SI) \leftarrow (SI) + 1$ and $(DI) \leftarrow (DI) + 1$

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing

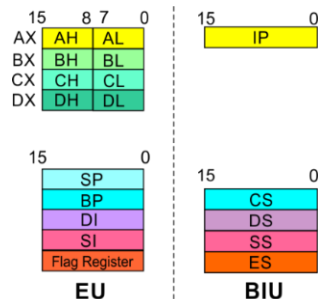
These addressing modes are used to access data from standard I/O mapped devices or ports.

In **direct port addressing mode**, an 8-bit port address is directly specified in the instruction.

Example: `IN AL, [09H]`

Operations: $\text{PORT}_{\text{addr}} = 09_{\text{H}}$
 $(\text{AL}) \leftarrow (\text{PORT})$

Content of port with address 09_{H} is moved to AL register



1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing

In this addressing mode, the effective address of a program instruction is specified relative to Instruction Pointer (IP) by an 8-bit signed displacement.

Example: JZ 0AH

Operations:

$000A_H \leftarrow 0A_H$ (sign extend)

If $ZF = 1$, then

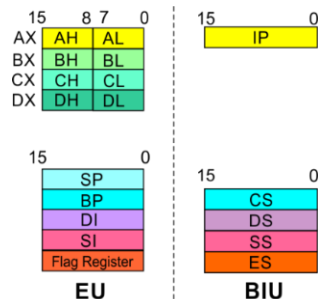
$EA = (IP) + 000A_H$

$BA = (CS) \times 16_{10}$

$MA = BA + EA$

If $ZF = 1$, then the program control jumps to new address calculated above.

If $ZF = 0$, then next instruction of the program is executed.



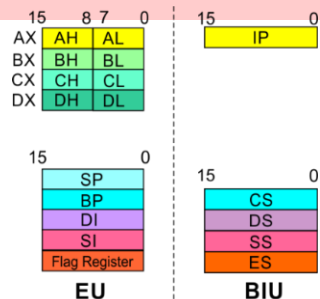
1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing

12. Implied Addressing

Instructions using this mode have no operands. The instruction itself will specify the data to be operated by the instruction.

Example: CLC

This clears the carry flag to zero.



INSTRUCTION SET

8086 supports 6 types of instructions.

- 1. Data Transfer Instructions**
- 2. Arithmetic Instructions**
- 3. Logical Instructions**
- 4. String manipulation Instructions**
- 5. Process Control Instructions**
- 6. Control Transfer Instructions**

1. Data Transfer Instructions

Instructions that are used to transfer data/ address in to registers, memory locations and I/O ports.

Generally involve two operands: Source operand and Destination operand of the same size.

Source: Register or a memory location or an immediate data
Destination : Register or a memory location.

The size should be a either a byte or a word.

A 8-bit data can only be moved to 8-bit register/ memory and a 16-bit data can be moved to 16-bit register/ memory.

Instruction Set

1. Data Transfer Instructions

Mnemonics: **MOV, XCHG, PUSH, POP, IN, OUT ...**

MOV reg2/ mem, reg1/ mem

MOV reg2, reg1
MOV mem, reg1
MOV reg2, mem

$(\text{reg2}) \leftarrow (\text{reg1})$
 $(\text{mem}) \leftarrow (\text{reg1})$
 $(\text{reg2}) \leftarrow (\text{mem})$

MOV reg/ mem, data

MOV reg, data
MOV mem, data

$(\text{reg}) \leftarrow \text{data}$
 $(\text{mem}) \leftarrow \text{data}$

XCHG reg2/ mem, reg1

XCHG reg2, reg1
XCHG mem, reg1

$(\text{reg2}) \leftrightarrow (\text{reg1})$
 $(\text{mem}) \leftrightarrow (\text{reg1})$

Instruction Set

1. Data Transfer Instructions

Mnemonics: **MOV, XCHG, PUSH, POP, IN, OUT ...**

PUSH reg16/ mem

PUSH reg16

$$\begin{aligned}(\text{SP}) &\leftarrow (\text{SP}) - 2 \\ \text{MA}_s &= (\text{SS}) \times 16_{10} + \text{SP} \\ (\text{MA}_s ; \text{MA}_s + 1) &\leftarrow (\text{reg16})\end{aligned}$$

PUSH mem

$$\begin{aligned}(\text{SP}) &\leftarrow (\text{SP}) - 2 \\ \text{MA}_s &= (\text{SS}) \times 16_{10} + \text{SP} \\ (\text{MA}_s ; \text{MA}_s + 1) &\leftarrow (\text{mem})\end{aligned}$$

POP reg16/ mem

POP reg16

$$\begin{aligned}\text{MA}_s &= (\text{SS}) \times 16_{10} + \text{SP} \\ (\text{reg16}) &\leftarrow (\text{MA}_s ; \text{MA}_s + 1) \\ (\text{SP}) &\leftarrow (\text{SP}) + 2\end{aligned}$$

POP mem

$$\begin{aligned}\text{MA}_s &= (\text{SS}) \times 16_{10} + \text{SP} \\ (\text{mem}) &\leftarrow (\text{MA}_s ; \text{MA}_s + 1) \\ (\text{SP}) &\leftarrow (\text{SP}) + 2\end{aligned}$$

Instruction Set

1. Data Transfer Instructions

Mnemonics: **MOV, XCHG, PUSH, POP, IN, OUT ...**

IN A, [DX]

IN AL, [DX]

$\text{PORT}_{\text{addr}} = (\text{DX})$
 $(\text{AL}) \leftarrow (\text{PORT})$

IN AX, [DX]

$\text{PORT}_{\text{addr}} = (\text{DX})$
 $(\text{AX}) \leftarrow (\text{PORT})$

IN A, addr8

IN AL, addr8

$(\text{AL}) \leftarrow (\text{addr8})$

IN AX, addr8

$(\text{AX}) \leftarrow (\text{addr8})$

OUT [DX], A

OUT [DX], AL

$\text{PORT}_{\text{addr}} = (\text{DX})$
 $(\text{PORT}) \leftarrow (\text{AL})$

OUT [DX], AX

$\text{PORT}_{\text{addr}} = (\text{DX})$
 $(\text{PORT}) \leftarrow (\text{AX})$

OUT addr8, A

OUT addr8, AL

$(\text{addr8}) \leftarrow (\text{AL})$

OUT addr8, AX

$(\text{addr8}) \leftarrow (\text{AX})$

2. Arithmetic Instructions

Mnemonics: **ADD**, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP...

ADD reg2/ mem, reg1/mem

ADD reg2, reg1
ADD reg2, mem
ADD mem, reg1

$(\text{reg2}) \leftarrow (\text{reg1}) + (\text{reg2})$
 $(\text{reg2}) \leftarrow (\text{reg2}) + (\text{mem})$
 $(\text{mem}) \leftarrow (\text{mem}) + (\text{reg1})$

ADD reg/mem, data

ADD reg, data
ADD mem, data

$(\text{reg}) \leftarrow (\text{reg}) + \text{data}$
 $(\text{mem}) \leftarrow (\text{mem}) + \text{data}$

ADD A, data

ADD AL, data8
ADD AX, data16

$(\text{AL}) \leftarrow (\text{AL}) + \text{data8}$
 $(\text{AX}) \leftarrow (\text{AX}) + \text{data16}$

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP...**

ADC reg2/ mem, reg1/mem

ADC reg2, reg1
ADC reg2, mem
ADC mem, reg1

$(\text{reg2}) \leftarrow (\text{reg1}) + (\text{reg2}) + \text{CF}$
 $(\text{reg2}) \leftarrow (\text{reg2}) + (\text{mem}) + \text{CF}$
 $(\text{mem}) \leftarrow (\text{mem}) + (\text{reg1}) + \text{CF}$

ADC reg/mem, data

ADC reg, data
ADC mem, data

$(\text{reg}) \leftarrow (\text{reg}) + \text{data} + \text{CF}$
 $(\text{mem}) \leftarrow (\text{mem}) + \text{data} + \text{CF}$

ADC A, data

ADC AL, data8
ADC AX, data16

$(\text{AL}) \leftarrow (\text{AL}) + \text{data8} + \text{CF}$
 $(\text{AX}) \leftarrow (\text{AX}) + \text{data16} + \text{CF}$

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP...**

SUB reg2/ mem, reg1/mem

SUB reg2, reg1
SUB reg2, mem
SUB mem, reg1

$(\text{reg2}) \leftarrow (\text{reg1}) - (\text{reg2})$
 $(\text{reg2}) \leftarrow (\text{reg2}) - (\text{mem})$
 $(\text{mem}) \leftarrow (\text{mem}) - (\text{reg1})$

SUB reg/mem, data

SUB reg, data
SUB mem, data

$(\text{reg}) \leftarrow (\text{reg}) - \text{data}$
 $(\text{mem}) \leftarrow (\text{mem}) - \text{data}$

SUB A, data

SUB AL, data8
SUB AX, data16

$(\text{AL}) \leftarrow (\text{AL}) - \text{data8}$
 $(\text{AX}) \leftarrow (\text{AX}) - \text{data16}$

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP...**

SBB reg2/ mem, reg1/mem

SBB reg2, reg1
SBB reg2, mem
SBB mem, reg1

$(\text{reg2}) \leftarrow (\text{reg1}) - (\text{reg2}) - \text{CF}$
 $(\text{reg2}) \leftarrow (\text{reg2}) - (\text{mem}) - \text{CF}$
 $(\text{mem}) \leftarrow (\text{mem}) - (\text{reg1}) - \text{CF}$

SBB reg/mem, data

SBB reg, data
SBB mem, data

$(\text{reg}) \leftarrow (\text{reg}) - \text{data} - \text{CF}$
 $(\text{mem}) \leftarrow (\text{mem}) - \text{data} - \text{CF}$

SBB A, data

SBB AL, data8
SBB AX, data16

$(\text{AL}) \leftarrow (\text{AL}) - \text{data8} - \text{CF}$
 $(\text{AX}) \leftarrow (\text{AX}) - \text{data16} - \text{CF}$

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP...**

INC reg/ mem

INC reg8

$(\text{reg8}) \leftarrow (\text{reg8}) + 1$

INC reg16

$(\text{reg16}) \leftarrow (\text{reg16}) + 1$

INC mem

$(\text{mem}) \leftarrow (\text{mem}) + 1$

DEC reg/ mem

DEC reg8

$(\text{reg8}) \leftarrow (\text{reg8}) - 1$

DEC reg16

$(\text{reg16}) \leftarrow (\text{reg16}) - 1$

DEC mem

$(\text{mem}) \leftarrow (\text{mem}) - 1$

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP...**

MUL reg/ mem (Unsigned)

MUL reg

For byte : $(AX) \leftarrow (AL) \times (\text{reg8})$

For word : $(DX)(AX) \leftarrow (AX) \times (\text{reg16})$

MUL mem

For byte : $(AX) \leftarrow (AL) \times (\text{mem8})$

For word : $(DX)(AX) \leftarrow (AX) \times (\text{mem16})$

IMUL reg/ mem (Signed)

IMUL reg

For byte : $(AX) \leftarrow (AL) \times (\text{reg8})$

For word : $(DX)(AX) \leftarrow (AX) \times (\text{reg16})$

IMUL mem

For byte : $(AX) \leftarrow (AX) \times (\text{mem8})$

For word : $(DX)(AX) \leftarrow (AX) \times (\text{mem16})$

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP...**

DIV reg/ mem

DIV reg

For 16-bit :- 8-bit :

(AL) \leftarrow (AX) :- (reg8) Quotient

(AH) \leftarrow (AX) MOD(reg8) Remainder

For 32-bit :- 16-bit :

(AX) \leftarrow (DX)(AX) :- (reg16) Quotient

(DX) \leftarrow (DX)(AX) MOD(reg16) Remainder

DIV mem

For 16-bit :- 8-bit :

(AL) \leftarrow (AX) :- (mem8) Quotient

(AH) \leftarrow (AX) MOD(mem8) Remainder

For 32-bit :- 16-bit :

(AX) \leftarrow (DX)(AX) :- (mem16) Quotient

(DX) \leftarrow (DX)(AX) MOD(mem16) Remainder

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP...**

IDIV reg/ mem**IDIV reg****For 16-bit :- 8-bit :**

(AL) \leftarrow (AX) :- (reg8) Quotient

(AH) \leftarrow (AX) MOD(reg8) Remainder

For 32-bit :- 16-bit :

(AX) \leftarrow (DX)(AX) :- (reg16) Quotient

(DX) \leftarrow (DX)(AX) MOD(reg16) Remainder

IDIV mem**For 16-bit :- 8-bit :**

(AL) \leftarrow (AX) :- (mem8) Quotient

(AH) \leftarrow (AX) MOD(mem8) Remainder

For 32-bit :- 16-bit :

(AX) \leftarrow (DX)(AX) :- (mem16) Quotient

(DX) \leftarrow (DX)(AX) MOD(mem16) Remainder

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP...**

CMP reg2/mem, reg1/ mem

CMP reg2, reg1

Modify flags \leftarrow (reg2) – (reg1)

If (reg2) > (reg1) then CF=0, ZF=0, SF=0

If (reg2) < (reg1) then CF=1, ZF=0, SF=1

If (reg2) = (reg1) then CF=0, ZF=1, SF=0

CMP reg2, mem

Modify flags \leftarrow (reg2) – (mem)

If (reg2) > (mem) then CF=0, ZF=0, SF=0

If (reg2) < (mem) then CF=1, ZF=0, SF=1

If (reg2) = (mem) then CF=0, ZF=1, SF=0

CMP mem, reg1

Modify flags \leftarrow (mem) – (reg1)

If (mem) > (reg1) then CF=0, ZF=0, SF=0

If (mem) < (reg1) then CF=1, ZF=0, SF=1

If (mem) = (reg1) then CF=0, ZF=1, SF=0

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP...**

CMP reg/mem, data

CMP reg, data

Modify flags \leftarrow (reg) - (data)

If (reg) > data then CF=0, ZF=0, SF=0

If (reg) < data then CF=1, ZF=0, SF=1

If (reg) = data then CF=0, ZF=1, SF=0

CMP mem, data

Modify flags \leftarrow (mem) - (data)

If (mem) > data then CF=0, ZF=0, SF=0

If (mem) < data then CF=1, ZF=0, SF=1

If (mem) = data then CF=0, ZF=1, SF=0

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP...**

CMP A, data

CMP AL, data8

Modify flags \leftarrow (AL) – data8

If (AL) > data8 then CF=0, ZF=0, SF=0

If (AL) < data8 then CF=1, ZF=0, SF=1

If (AL) = data8 then CF=0, ZF=1, SF=0

CMP AX, data16

Modify flags \leftarrow (AX) – data16

If (AX) > data16 then CF=0, ZF=0, SF=0

If (mem) < data16 then CF=1, ZF=0, SF=1

If (mem) = data16 then CF=0, ZF=1, SF=0

3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...**

AND A, data AND AL, data8	$(AL) \leftarrow (AL) \& \text{data8}$
AND AX, data16	$(AX) \leftarrow (AX) \& \text{data16}$

AND reg/mem, data AND reg, data	$(\text{reg}) \leftarrow (\text{reg}) \& \text{data}$
AND mem, data	$(\text{mem}) \leftarrow (\text{mem}) \& \text{data}$

3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...**

OR reg2/mem, reg1/mem OR reg2, reg1 OR reg2, mem OR mem, reg1	$(reg2) \leftarrow (reg2) \mid (reg1)$ $(reg2) \leftarrow (reg2) \mid (mem)$ $(mem) \leftarrow (mem) \mid (reg1)$
OR reg/mem, data OR reg, data OR mem, data	$(reg) \leftarrow (reg) \mid data$ $(mem) \leftarrow (mem) \mid data$
OR A, data OR AL, data8 OR AX, data16	$(AL) \leftarrow (AL) \mid data8$ $(AX) \leftarrow (AX) \mid data16$

3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...**

XOR reg2/mem, reg1/mem XOR reg2, reg1 XOR reg2, mem XOR mem, reg1	$(reg2) \leftarrow (reg2) \wedge (reg1)$ $(reg2) \leftarrow (reg2) \wedge (mem)$ $(mem) \leftarrow (mem) \wedge (reg1)$
XOR reg/mem, data XOR reg, data XOR mem, data	$(reg) \leftarrow (reg) \wedge data$ $(mem) \leftarrow (mem) \wedge data$
XOR A, data XOR AL, data8 XOR AX, data16	$(AL) \leftarrow (AL) \wedge data8$ $(AX) \leftarrow (AX) \wedge data16$

3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...**

TEST reg2/mem, reg1/mem TEST reg2, reg1 TEST reg2, mem TEST mem, reg1	Modify flags \leftarrow (reg2) & (reg1) Modify flags \leftarrow (reg2) & (mem) Modify flags \leftarrow (mem) & (reg1)
TEST reg/mem, data TEST reg, data TEST mem, data	Modify flags \leftarrow (reg) & data Modify flags \leftarrow (mem) & data
TEST A, data TEST AL, data8 TEST AX, data16	Modify flags \leftarrow (AL) & data8 Modify flags \leftarrow (AX) & data16

3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...**

SHR reg/mem

SHR reg

i) SHR reg, 1

ii) SHR reg, CL

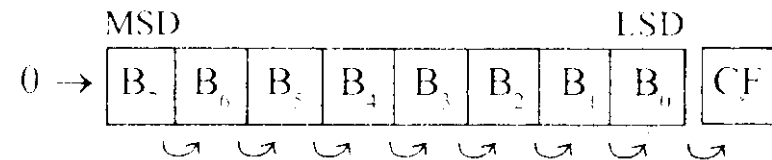
SHR mem

i) SHR mem, 1

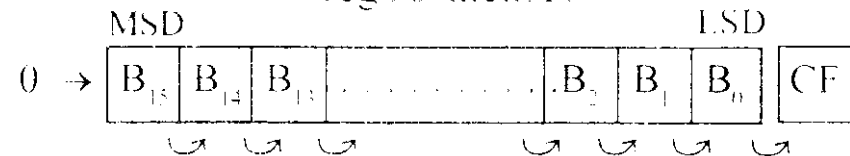
ii) SHR mem, CL

$$CF \leftarrow B_{\text{LSD}} ; B_n \leftarrow B_{n+1} ; B_{\text{MSD}} \leftarrow 0$$

reg 8 / mem 8



reg 16 / mem 16



3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...**

SHL reg/mem or SAL reg/mem

SHL reg or SAL reg

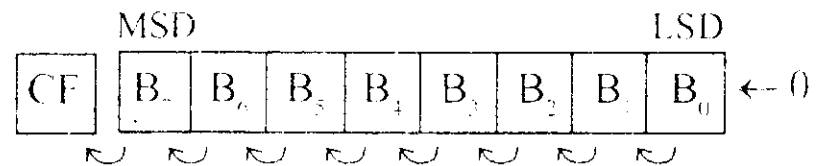
- i) SHL reg, 1 or SAL reg, 1
- ii) SHL reg, CL or SAL reg, CL

SHL mem or SAL mem

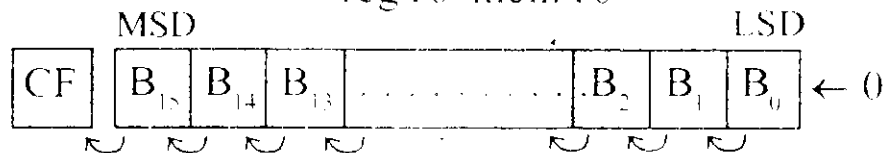
- i) SHL mem, 1 or SAL mem, 1
- ii) SHL mem, CL or SAL mem, CL

$$CF \leftarrow B_{MSD} ; B_{n+1} \leftarrow B_n ; B_{LSD} \leftarrow 0$$

reg 8 / mem 8



reg 16 / mem 16



3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL, ROR, ROL ...**

RCR reg/mem

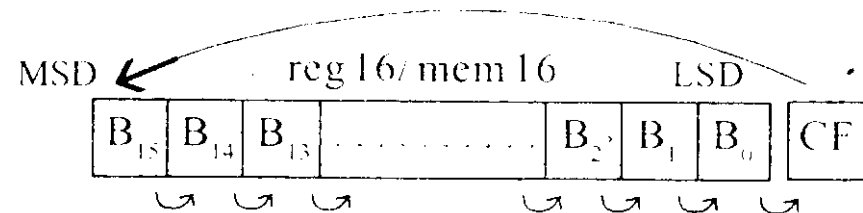
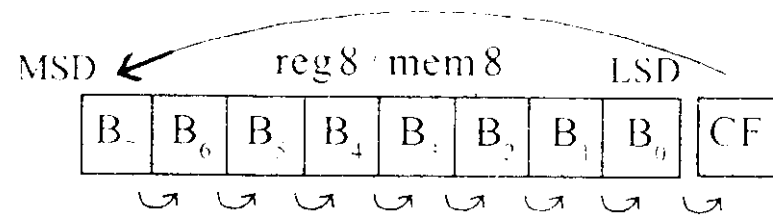
RCR reg

- i) RCR reg, 1
- ii) RCR reg, CL

RCR mem

- i) RCR mem, 1
- ii) RCR mem, CL

$$B_n \leftarrow B_{n-1} ; B_{\text{MSD}} \leftarrow \text{CF} ; \text{CF} \leftarrow B_{\text{LSD}}$$



3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...**

ROL reg/mem

ROL reg

i) ROL reg, 1

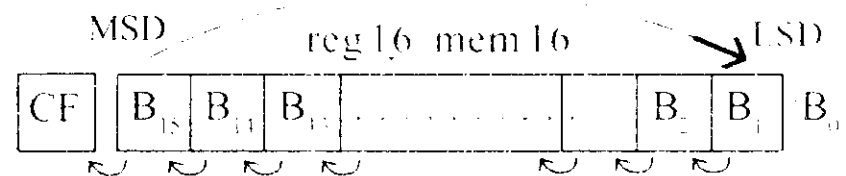
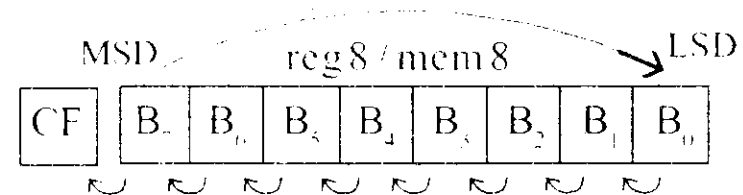
ii) ROL reg, CL

ROL mem

i) ROL mem, 1

ii) ROL mem, CL

$$B_{n+1} \leftarrow B_n ; CF \leftarrow B_{MSD} ; B_{LSD} \leftarrow B_{MSD}$$



4. String Manipulation Instructions

- ❑ String : Sequence of bytes or words
- ❑ 8086 instruction set includes instruction for string movement, comparison, scan, load and store.
- ❑ REP instruction prefix : used to repeat execution of string instructions
- ❑ String instructions end with S or SB or SW.
S represents string, SB string byte and SW string word.
- ❑ Offset or effective address of the source operand is stored in SI register and that of the destination operand is stored in DI register.
- ❑ Depending on the status of DF, SI and DI registers are automatically updated.
- ❑ $DF = 0 \Rightarrow SI$ and DI are incremented by 1 for byte and 2 for word.
- ❑ $DF = 1 \Rightarrow SI$ and DI are decremented by 1 for byte and 2 for word.

4. String Manipulation Instructions

Mnemonics: **REP, MOVS, CMPS, SCAS, LODS, STOS**

REP

REPZ/ REPE

**(Repeat CMPS or SCAS until
ZF = 0)**

**While $CX \neq 0$ and $ZF = 1$, repeat execution of
string instruction and
 $(CX) \leftarrow (CX) - 1$**

REPNZ/ REPNE

**(Repeat CMPS or SCAS until
ZF = 1)**

**While $CX \neq 0$ and $ZF = 0$, repeat execution of
string instruction and
 $(CX) \leftarrow (CX) - 1$**

4. String Manipulation Instructions

Mnemonics: **REP, MOVSB, CMPS, SCAS, LODS, STOS**

MOVSB

MOVSB

$$\begin{aligned} \text{MA} &= (\text{DS}) \times 16_{10} + (\text{SI}) \\ \text{MA}_E &= (\text{ES}) \times 16_{10} + (\text{DI}) \end{aligned}$$

$$(\text{MA}_E) \leftarrow (\text{MA})$$

If $\text{DF} = 0$, then $(\text{DI}) \leftarrow (\text{DI}) + 1$; $(\text{SI}) \leftarrow (\text{SI}) + 1$

If $\text{DF} = 1$, then $(\text{DI}) \leftarrow (\text{DI}) - 1$; $(\text{SI}) \leftarrow (\text{SI}) - 1$

MOVSW

$$\begin{aligned} \text{MA} &= (\text{DS}) \times 16_{10} + (\text{SI}) \\ \text{MA}_E &= (\text{ES}) \times 16_{10} + (\text{DI}) \end{aligned}$$

$$(\text{MA}_E ; \text{MA}_E + 1) \leftarrow (\text{MA}; \text{MA} + 1)$$

If $\text{DF} = 0$, then $(\text{DI}) \leftarrow (\text{DI}) + 2$; $(\text{SI}) \leftarrow (\text{SI}) + 2$

If $\text{DF} = 1$, then $(\text{DI}) \leftarrow (\text{DI}) - 2$; $(\text{SI}) \leftarrow (\text{SI}) - 2$

4. String Manipulation Instructions

Mnemonics: **REP, MOVS, CMPS, SCAS, LODS, STOS**

Compare two string byte or string word

CMPS

CMPSB

CMPSW

$MA = (DS) \times 16_{10} + (SI)$
 $MA_E = (ES) \times 16_{10} + (DI)$

Modify flags $\leftarrow (MA) - (MA_E)$

If $(MA) > (MA_E)$, then $CF = 0$; $ZF = 0$; $SF = 0$

If $(MA) < (MA_E)$, then $CF = 1$; $ZF = 0$; $SF = 1$

If $(MA) = (MA_E)$, then $CF = 0$; $ZF = 1$; $SF = 0$

For byte operation

If $DF = 0$, then $(DI) \leftarrow (DI) + 1$; $(SI) \leftarrow (SI) + 1$

If $DF = 1$, then $(DI) \leftarrow (DI) - 1$; $(SI) \leftarrow (SI) - 1$

For word operation

If $DF = 0$, then $(DI) \leftarrow (DI) + 2$; $(SI) \leftarrow (SI) + 2$

If $DF = 1$, then $(DI) \leftarrow (DI) - 2$; $(SI) \leftarrow (SI) - 2$

4. String Manipulation Instructions

Mnemonics: **REP, MOVS, CMPS, SCAS, LODS, STOS**

Scan (compare) a string byte or word with accumulator

SCAS

SCASB

$MA_E = (ES) \times 16_{10} + (DI)$
Modify flags $\leftarrow (AL) - (MA_E)$

If $(AL) > (MA_E)$, then $CF = 0$; $ZF = 0$; $SF = 0$

If $(AL) < (MA_E)$, then $CF = 1$; $ZF = 0$; $SF = 1$

If $(AL) = (MA_E)$, then $CF = 0$; $ZF = 1$; $SF = 0$

If $DF = 0$, then $(DI) \leftarrow (DI) + 1$

If $DF = 1$, then $(DI) \leftarrow (DI) - 1$

SCASW

$MA_E = (ES) \times 16_{10} + (DI)$
Modify flags $\leftarrow (AX) - (MA_E)$

If $(AX) > (MA_E ; MA_E + 1)$, then $CF = 0$; $ZF = 0$; $SF = 0$

If $(AX) < (MA_E ; MA_E + 1)$, then $CF = 1$; $ZF = 0$; $SF = 1$

If $(AX) = (MA_E ; MA_E + 1)$, then $CF = 0$; $ZF = 1$; $SF = 0$

If $DF = 0$, then $(DI) \leftarrow (DI) + 2$

If $DF = 1$, then $(DI) \leftarrow (DI) - 2$

4. String Manipulation Instructions

Mnemonics: **REP, MOVS, CMPS, SCAS, LODS, STOS**

Load string byte in to AL or string word in to AX

LODS

LODSB

$MA = (DS) \times 16_{10} + (SI)$
 $(AL) \leftarrow (MA)$

If $DF = 0$, then $(SI) \leftarrow (SI) + 1$
If $DF = 1$, then $(SI) \leftarrow (SI) - 1$

LODSW

$MA = (DS) \times 16_{10} + (SI)$
 $(AX) \leftarrow (MA ; MA + 1)$

If $DF = 0$, then $(SI) \leftarrow (SI) + 2$
If $DF = 1$, then $(SI) \leftarrow (SI) - 2$

4. String Manipulation Instructions

Mnemonics: **REP, MOVS, CMPS, SCAS, LODS, STOS**

Store byte from AL or word from AX in to memory location

STOS

STOSB

$MA_E = (ES) \times 16_{10} + (DI)$
 $(MA_E) \leftarrow (AL)$

If $DF = 0$, then $(DI) \leftarrow (DI) + 1$
If $DF = 1$, then $(DI) \leftarrow (DI) - 1$

STOSW

$MA_E = (ES) \times 16_{10} + (DI)$
 $(MA_E ; MA_E + 1) \leftarrow (AX)$

If $DF = 0$, then $(DI) \leftarrow (DI) + 2$
If $DF = 1$, then $(DI) \leftarrow (DI) - 2$

5. Processor Control Instructions

Mnemonics	Explanation
STC	Set CF $\leftarrow 1$
CLC	Clear CF $\leftarrow 0$
CMC	Complement carry CF $\leftarrow \text{CF}'$
STD	Set direction flag DF $\leftarrow 1$
CLD	Clear direction flag DF $\leftarrow 0$
STI	Set interrupt enable flag IF $\leftarrow 1$
CLI	Clear interrupt enable flag IF $\leftarrow 0$
NOP	No operation
HLT	Halt after interrupt is set
WAIT	Wait for TEST pin active
ESC opcode mem/ reg	Used to pass instruction to a coprocessor which shares the address and data bus with the 8086
LOCK	Lock bus during next instruction

6. Control Transfer Instructions

- Transfer the control to a specific destination or target instruction
- Do not affect flags

□ 8086 Unconditional transfers

Mnemonics	Explanation
CALL reg/ mem/ disp16	Call subroutine
RET	Return from subroutine
JMP reg/ mem/ disp8/ disp16	Unconditional jump

6. Control Transfer Instructions

- ❑ **8086 signed conditional branch instructions**
 - ❑ **8086 unsigned conditional branch instructions**
-
- **Checks flags**
 - **If conditions are true, the program control is transferred to the new memory location in the same segment by modifying the content of IP**

6. Control Transfer Instructions

❑ 8086 signed conditional branch instructions

Name	Alternate name
JE disp8 Jump if equal	JZ disp8 Jump if result is 0
JNE disp8 Jump if not equal	JNZ disp8 Jump if not zero
JG disp8 Jump if greater	JNLE disp8 Jump if not less or equal
JGE disp8 Jump if greater than or equal	JNL disp8 Jump if not less
JL disp8 Jump if less than	JNGE disp8 Jump if not greater than or equal
JLE disp8 Jump if less than or equal	JNG disp8 Jump if not greater

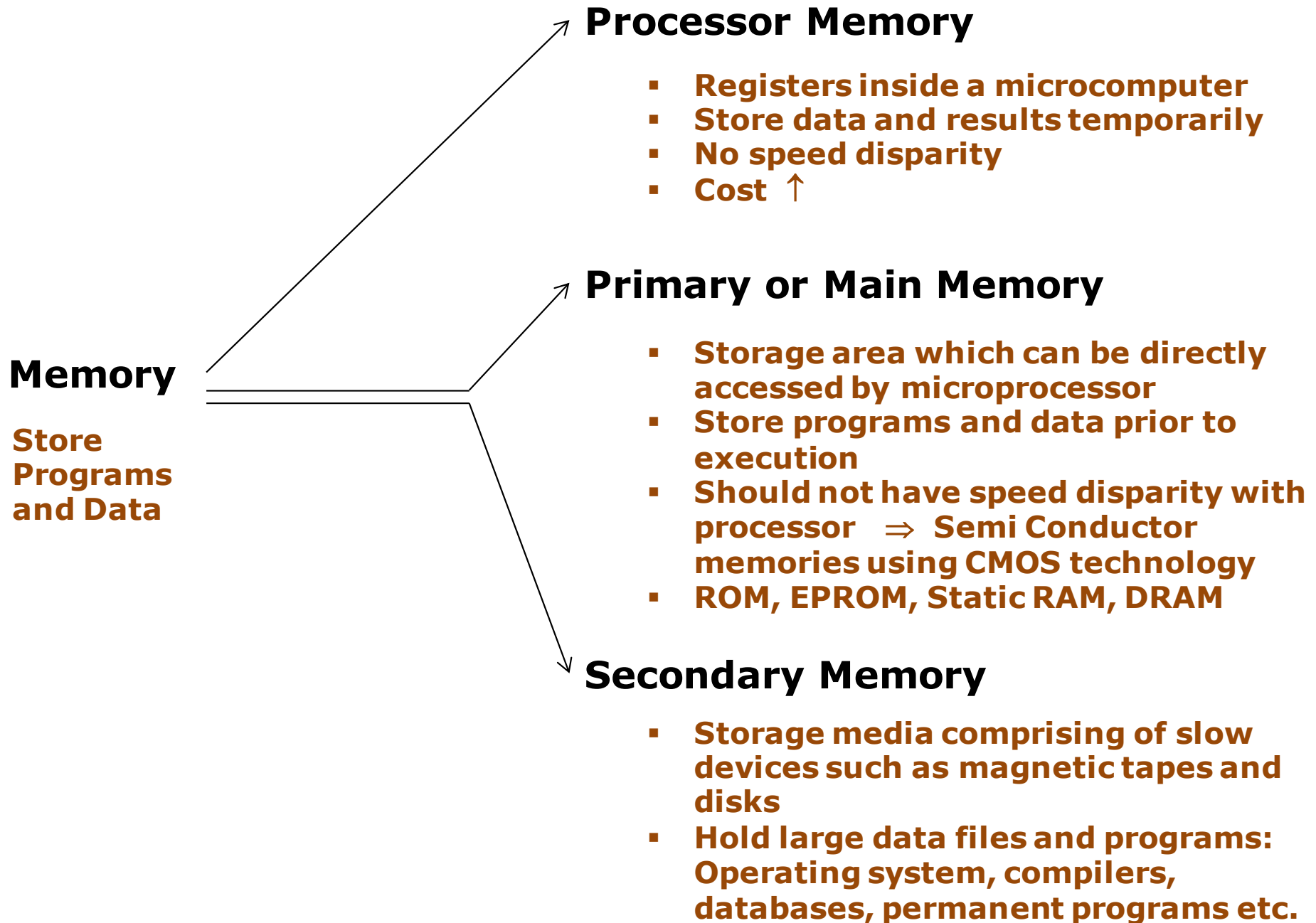
❑ 8086 unsigned conditional branch instructions

Name	Alternate name
JE disp8 Jump if equal	JZ disp8 Jump if result is 0
JNE disp8 Jump if not equal	JNZ disp8 Jump if not zero
JA disp8 Jump if above	JNBE disp8 Jump if not below or equal
JAE disp8 Jump if above or equal	JNB disp8 Jump if not below
JB disp8 Jump if below	JNAE disp8 Jump if not above or equal
JBE disp8 Jump if below or equal	JNA disp8 Jump if not above

6. Control Transfer Instructions

- ❑ 8086 conditional branch instructions affecting individual flags

Mnemonics	Explanation
JC disp8	Jump if CF = 1
JNC disp8	Jump if CF = 0
JP disp8	Jump if PF = 1
JNP disp8	Jump if PF = 0
JO disp8	Jump if OF = 1
JNO disp8	Jump if OF = 0
JS disp8	Jump if SF = 1
JNS disp8	Jump if SF = 0
JZ disp8	Jump if result is zero, i.e, Z = 1
JNZ disp8	Jump if result is not zero, i.e, Z = 1

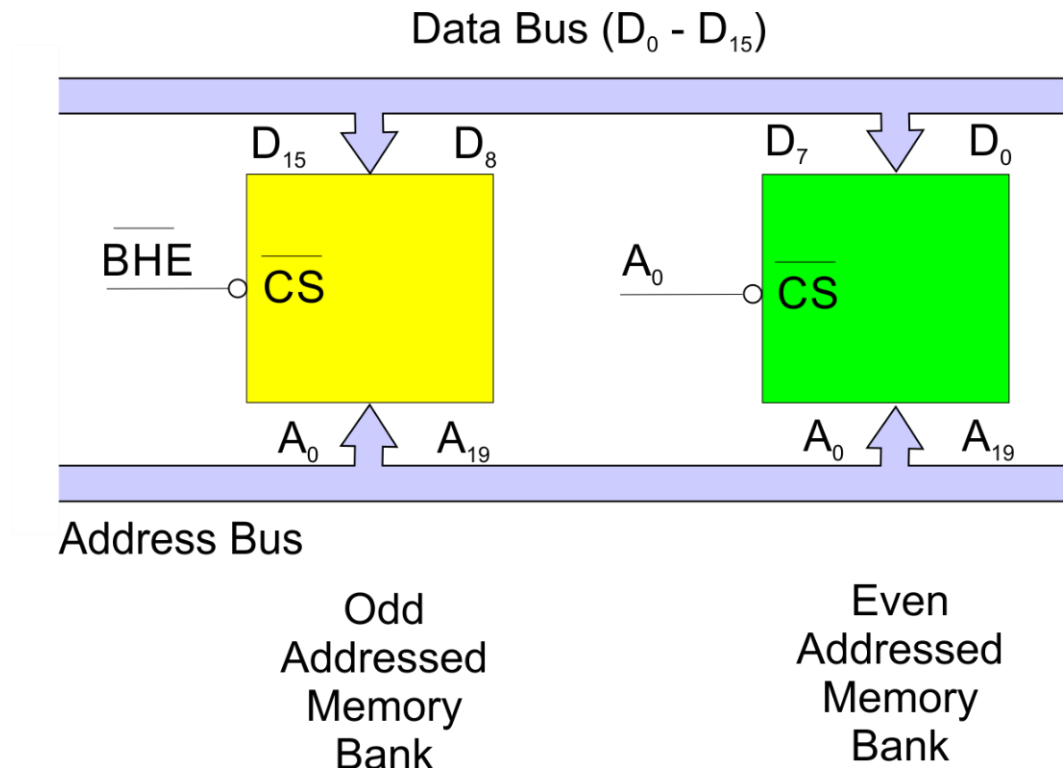


Physical Memory Organization

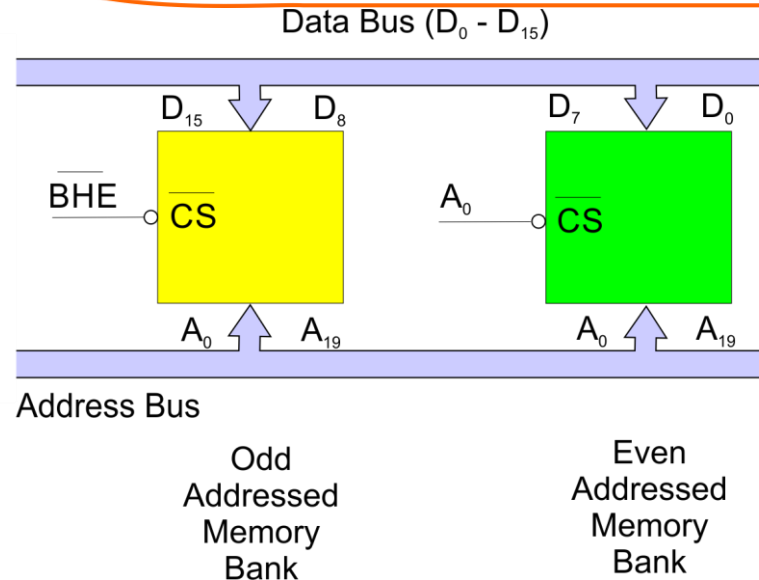
- 1MB memory is physically organized as an odd bank and an even bank, each of 512 KB
- Byte data with an even address is transferred on D7-D0
- Byte data with an odd address is transferred on D15-D8
- FFFF0h to FFFFFh are reserved for initialization of program & I/O process, while 000000h to 003FFh (1KB for 256 interrupt vectors) are reserved for interrupt vector table.

Memory organization in 8086

- **Memory IC's : Byte oriented**
- **8086 : 16-bit**
- **Word : Stored by two consecutive memory locations; for LSB and MSB**
- **Address of word : Address of LSB**
- **Bank 0 : $A_0 = 0 \Rightarrow$ Even addressed memory bank**
- **Bank 1 : $\overline{BHE} = 0 \Rightarrow$ Odd addressed memory bank**



Memory organization in 8086



	Operation	\overline{BHE}	A ₀	Data Lines Used
1	Read/ Write byte at an even address	1	0	D ₇ - D ₀
2	Read/ Write byte at an odd address	0	1	D ₁₅ - D ₈
3	Read/ Write word at an even address	0	0	D ₁₅ - D ₀
4	Read/ Write word at an odd address	0	1	D ₁₅ - D ₈ in first operation byte from odd bank is transferred
		1	0	D ₇ - D ₀ in second operation byte from even bank is transferred

Comparison between 8085 & 8086 Microprocessor

Size – 8085 is 8-bit microprocessor, whereas 8086 is 16-bit microprocessor.

Address Bus – 8085 has 16-bit address bus while 8086 has 20-bit address bus.

Memory – 8085 can access up to 64Kb, whereas 8086 can access up to 1 Mb of memory.

Instruction – 8085 doesn't have an instruction queue, whereas 8086 has an instruction queue.

Pipelining – 8085 doesn't support a pipelined architecture while 8086 supports a pipelined architecture.

I/O – 8085 can address $2^8 = 256$ I/O's, whereas 8086 can access $2^{16} = 65,536$ I/O's.

Cost – The cost of 8085 is low whereas that of 8086 is high.